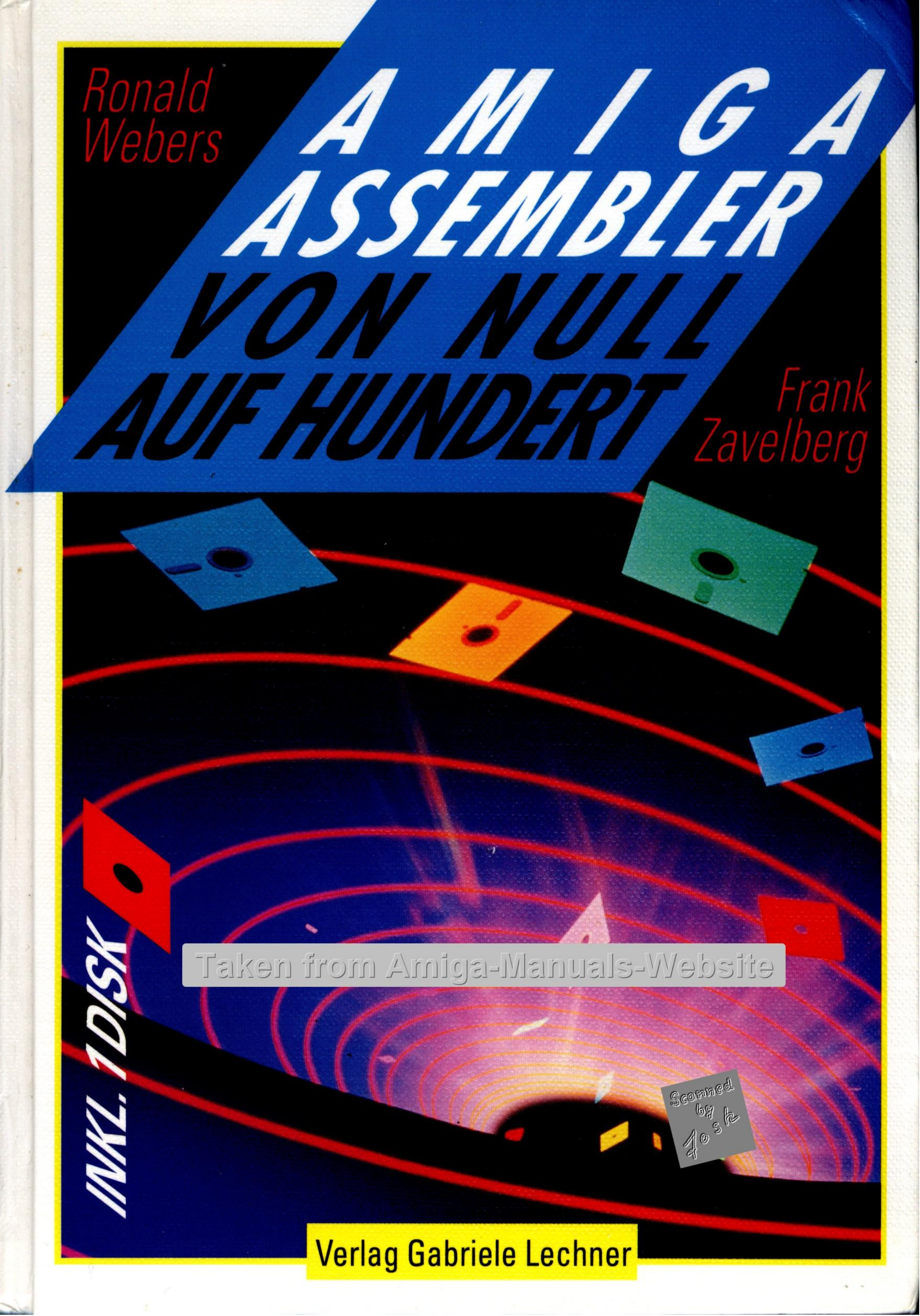

Ronald
Webers
AMIGA
ASSEMBLER
VON NULL
AUF HUNDERT
Frank
Zavelberg
INKL. 1 DISK
Taken from Amiga-Manuals-Website
Verlag Gabriele Lechner

AMIGA ASSEMBLER
VON NULL AUF HUNDERT

AMIGA ASSEMBLER
VON NULL AUF HUNDERT

Ronald Webers
Frank Zavelberg

Verlag Gabriele Lechner

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Webers, Ronald:
Amiga Assembler - von Null auf Hundert / Ronald Webers ;
Frank Zavelberg. - 1. Aufl. - München : Lechner, 1993
ISBN 3-926858-40-0
NE: Zavelberg, Frank:

ISBN 3-926858-40-0
1. Auflage 1993

Einbandgestaltung:	Young Sue Niedermeier
Titelfoto:	IMAGE BANK 215682 Steven Hunt
Druck:	WB-Druck GmbH & Co. Rieden am Forgensee
Autoren:	Ronald Webers Frank Zavelberg

Inhaltsverzeichnis

Vorwort

Der Amiga ist aufgrund seines erschwinglichen Preises und seiner bestechenden Grafik- und Soundfähigkeiten in kürzester Zeit sehr bekannt geworden. Dies veranlaßte viele Interessenten, sich mit der Programmierung des Amiga zu beschäftigen. Man könnte auf die zahlreichen höheren Programmiersprachen zurückgreifen, doch der umfangreiche Wortschatz des MC68000 ermöglicht es, auch in Assembler komfortabel zu programmieren.

Vergleicht man den MC68000-Prozessor mit anderen bekannten Typen, wie dem 6502 (der im C-64 seinen Dienst tut) oder dem 8088-Reihe von Intel, so stellt man fest, daß der Amiga-Prozessor wesentlich mehr Möglichkeiten bietet. Beim Vergleich mit dem 6502 wird das kaum verwundern, doch sogar die Intel-Prozessoren können sich vom MC68000 noch eine Scheibe abschneiden.

Wer Assembler programmieren möchte, muß nicht nur die Sprache selber, also die Befehle und Anwendungsvorschriften (Syntax) des MC68000 kennen, sondern sich auch mit dem System des Proszessors auskennen. Die Sprache selber wird in diesem Buch in den ersten drei Kapiteln behandelt. Fast alles, was Sie dort lernen, können Sie auch auf andere Systemen anwenden, die den MC68000 besitzen.

Der Abschnitt Systemprogrammierung wird den weitaus größeren Teil des Buches (Kapitel 4-12) einnehmen. Er ist Amiga-spezifisch, die dort erworbenen Kenntnisse sind also nur auf dem Amiga einsetzbar.

Die ersten drei Kapitel sollen beim Einstieg in Assembler helfen. Hierzu werden die Grundbegriffe erklärt, und es wird auf substantielle Routinen bei Schleifenkonstruktionen eingegangen. Vom vierten bis achten Kapitel erhält man Einsicht in die fünf wichtigsten Libraries des Amiga. Das Kapitel 9 beschäftigt sich mit der Konstruktion von eigenen Libraries.

Kapitel (10) behandelt die Devices, die beim Amiga eine vergleichbare Stellung wie die Libraries einnehmen. Es wird auf die wichtigsten vier Devices eingegangen und anhand von zahlreichen Beispielprogrammen deren Anwendung demonstriert. In Kapitel 11 wird die Konstruktion eines eigenen Devices dokumentiert.

Zum Schluß wird in Kapitel 12 die Programmierung der DOS-Library und der Aufbau des AmigaDOS vertieft.

Auf den letzten Seiten finden sich zahlreiche Anhänge, in denen alle wichtigen Informationen zusammengefaßt sind. Sie sollen dieses Buch auch als Nachschlagewerk nutzbar machen.

Bevor wir loslegen, möchten wir noch folgenden Personen danen: dem Techniklehrer Herrn Knut Reinhardt vom Städtischen Gymnasium Rheinbach für die zur Verfügung gestellten Computer, Thorsten Jansen (die Thorsten-Post) für immer (nicht immer, aber immer öfter) pünktliche Überlieferung unserer Disketten, und allen anderen, die zur Entstehung dieses Buches beigetragen haben.

Viel Spaß beim Lesen und Programmieren wünschen Ihnen

Ronald Webers Frank Zavelberg

Kapitel 1
Einleitung

Was ist Assembler?

Wann und warum benutzt man Assembler?

Was wird zur Programmierung benötigt?

Aufbau eines Computers

Die verschiedenen Zahlensysteme

Datentypen: Bytes, Worte und Langworte

Der ASCII-Code

1.1 Was ist Assembler?

Dieses Kapitel soll Ihnen Grundsätzliches über den Assembler vermitteln. Unrecht haben nämlich mit Sicherheit die Leute, die behaupten, Assembler sei eine Sprache nur für Programmier-Profis. Auch in Assembler wird nur mit Wasser gekocht, und man kann sie lernen wie jede andere Programmiersprache.

Damit kommen wir zu der Frage, was denn die Besonderheiten von Assembler und die Unterschiede zwischen den Programmiersprachen sind.

1.1.1 Unterschiede Assembler-Hochsprachen

Vielleicht haben Sie schon Erfahrung in Sprachen wie BASIC oder PASCAL. Diese, wie auch die meisten anderen Sprachen, nennt man "Hochsprachen". Eine solche Hochsprache zeichnet aus, daß viele "einfache" Befehle im Computer letztendlich ziemlich komplexe Vorgänge auslösen. Der PRINT-Befehl in BASIC beispielsweise, ist von der internen Ausführung her recht kompliziert: Der auszugebende Text muß anhand des eingestellten Zeichensatzes in Grafikpunkte umgerechnet werden und dann an der richtigen Stelle in den Teil des Speichers geschrieben werden, der vom Computer auf dem Bildschirm dargestellt wird. Also eine ganze Menge Arbeit, die der BASIC-Interpreter für den Programmierer unsichtbar leistet.

Wenn wir uns nun auf die Stufe der Assembler-Programmierung begeben, müssen wir für alle diese Dinge, die uns BASIC schon zur Verfügung stellt, selbst sorgen. Das heißt natürlich nicht, daß Sie im Falle der Textausgabe selbst die richtigen Grafikpunkte auf dem Bildschirm ausrechnen müssen, dafür stellt der Computer vorgefertigte Programmteile (genannt "Routinen") zur Verfügung. Aber Sie müssen z.B. die Positionen der Texte auf dem Bildschirm etc. selbst überwachen.

1.1.2 Von Assembler zur Maschinensprache

Wie in Hochsprachen gibt es auch in Assembler viele Befehle. Der wohl am häufigsten gebrauchte Befehl in Assembler ist der MOVE-(Verschiebe-)Befehl. Wenn man z.B. schreibt

```
move    2,4
```

verschiebt der Computer das, was an der Stelle 2 im Speicher steht, an die Stelle 4.

Nun gibt es da aber ein kleines Problem: Der Computer ist dumm. Er ist so dumm, daß er nur die Ziffern 0 und 1 kennt. Das kommt daher, daß ein Computer auf einem System von Schaltkreisen basiert, die nur zwei Zustände annehmen können: Strom fließt oder Strom fließt nicht, also 0 oder 1. Dies bezeichnet man als "Dualsystem". Zahlen dieses Systems kann man auch in andere Systeme, z.B. in unser "normales" Zehner-(Dezimal-) System umrechnen. Wie das geht, werden wir später noch sehen.

Jedem Assembler-Befehl wird eine Nummer zugeordnet. Angenommen, der MOVE-Befehl hätte die (Dual-)Nummer 1111, dann sähe unser Befehl von vorhin so aus:

1111 (die Nummer des Befehls)
0010 (Dualzahl für 2)
0100 (Dualzahl für 4)

Man kann sich also ein Assembler-Programm auf der untersten Stufe als lange Kette aus Nullen und Einsen denken:

111100100100 ...

Diese Darstellung des Programmes nennt man Maschinensprache. Man darf also Maschinensprache nicht mit Assembler verwechseln.

Natürlich wäre es äußerst unpraktisch, ein Programm direkt in Maschinensprache zu schreiben. Wie gesagt stellt diese 010101-Folge die für den Computer übersetzte, direkt verständliche Form eines Assembler-Programms dar. Wir werden unsere Programme selbstverständlich in der "Befehlsform" (move 2,4) schreiben, die Übersetzung in den 0101-Code übernimmt ein entsprechendes Programm (das günstigerweise auch "Assembler" heißt).

1.2 Wann und warum Assembler?

Sprachen wie BASIC, so einfach sie auch zu programmieren sein mögen, haben einen ganz entscheidenden Nachteil: Es sind sog. Interpretersprachen, d.h. jeder Befehl, den Sie schreiben, muß während der Programmausführung interpretiert, also in Maschinensprache übersetzt werden, da der Computer nur diese direkt versteht. Wenn also in einer Schleife ein Befehl 100 mal ausgeführt wird, muß er auch 100 mal übersetzt werden. Diesen Nachteil kennt Assembler nicht. Hier sind die Befehle quasi schon übersetzt, was einen enormen Geschwindigkeitsunterschied ausmachen kann.

Es gibt noch eine andere Art von Programmiersprachen: die sog. Compilersprachen. Hier schreibt man auch ein Quellprogramm, ähnlich wie in BASIC, nur wird dieses nicht während der Laufzeit übersetzt, sondern es wird compiliert, d.h. das

Quellprogramm in der Hochsprache wird vor der ersten Benutzung einmal in ein Maschinenprogramm übersetzt. Damit wird natürlich eine Geschwindigkeitssteigerung erreicht, aber ein Compiler muß ziemlich allgemein gehaltene Übersetzungs-Verfahren verwenden, wo hingegen ein Assemblerprogrammierer seine Programme den jeweiligen Aufgaben genau anpassen kann.

Darüber hinaus bietet Assembler noch einen weiteren Vorteil: Hochsprachen können, da sie im Prinzip nicht an einen bestimmten Computer gebunden sind, nicht vollkommen auf die Möglichkeiten des jeweiligen Systems eingehen. Assembler hingegen, befindet sich sozusagen "direkt an der Quelle". In Assembler stehen Ihnen alle Möglichkeiten des Computers offen - Sie müssen sie nur zu nutzen wissen (und dafür ist ja dieses Buch da).

Man kann also sagen, daß Assembler immer dann eingesetzt werden sollte, wenn es um zeitkritische Probleme oder um volle Ausnutzung der Möglichkeiten geht. Selten werden komplette Programme wie Textverarbeitungen o.Ä. in Assembler geschrieben. Meist greift man auf die Möglichkeit zurück, Assembler und Hochsprache zu mischen, wodurch die Vorteile beider Sprachen verbunden werden können.

1.3 Was wird zur Assemblerprogrammierung benötigt?

Die bekanntesten Assembler-Pakete für den Amiga sind der Devpac und der Seka. Dieser Kurs ist auf den Devpac ausgerichtet, die Programme können jedoch auch problemlos auf den Seka oder andere Assembler umgeschrieben werden, da sie sich nur bei einigen wenigen Spezial-Befehlen unterscheiden. Sollten bei bestimmten Befehlen Probleme auftauchen, wird Ihnen ein Blick ins Handbuch des Assemblers bestimmt weiterhelfen. Im folgenden wollen wir die Funktion der einzelnen Teile, die zu einem Assembler gehören, kurz beschreiben. Wir werden hier allerdings auf eine genaue Anleitung, wie ein Assemblerprogramm einzugeben und zu übersetzen ist, verzichten. Dieser Ablauf ist nämlich bei jedem Assembler anders, und wird ausführlich im zugehörigen Handbuch beschrieben.

1.3.1 Der Editor

Der Editor dient zur Eingabe und Überarbeitung Ihres Programmes. Viele Assemblerpakete (wie z.B. der Devpac und der Seka) besitzen einen integrierten Editor. Falls Ihr Assembler keinen besitzt oder er Ihnen zu schlecht erscheint, können Sie auch jeden anderen Editor, der Texte im ASCII-Format (d.h. reinen Text, ohne Kommandos wie Fett, Kursiv usw.) abspeichern kann, verwenden.

Der Editor sollte über Funktionen zum Bewegen und Kopieren von Blöcken verfügen, da Sie Ihre Programme möglicherweise des öfteren umgestalten oder umordnen müssen. Auch wäre eine Goto-Zeile-Funktion von Nutzen, da der Assembler Fehlermeldungen in der Regel mit Zeilennummer ausgibt, ein Assemblerprogramm aber ohne Zeilennummern eingegeben wird.

Wenn Sie einen nicht-Assembler-integrierten Editor benutzen, müssen Sie das Programm abspeichern und dem Assembler (siehe nächster Abschnitt) den Dateinamen mitteilen.

1.3.2 Der Assembler

Dieses Programm, das sinnigerweise genauso heißt wie die Programmiersprache, ist für die Übersetzung Ihres Programms, des Quelltextes (auch Sourcecode genannt), zuständig. Der Assembler erzeugt entweder die lauffähige Version Ihres Programms auf Diskette oder er ist (wie der Devpac und Seka) in der Lage, direkt in den Speicher zu assemblieren. Letzteres hat den Vorteil, daß das Programm schnell zum Austesten bereit ist und erst abgespeichert werden muß, wenn alles funktioniert.

Die meisten Assembler bieten eine Funktion, die sich "Include" nennt. Das bedeutet, es können andere, auf Diskette befindliche, Quelltext-Teile während der Assemblierung per Befehl in einen Programmtext eingebunden werden. Der Seka kennt diesen Befehl nicht, aber er kann über das R-Kommando Textteile (vor der Assemblierung) dazuladen. Zu diesem und anderen Features werden wir in späteren Kapiteln noch kommen.

1.3.3 Der Debugger

Vielleicht gehören Sie zu den Glücklichen, deren Programme immer auf Anhieb funktionieren, dann werden Sie einen Debugger kaum brauchen. Falls nicht, kann er Ihnen bei der Fehlersuche recht hilfreich sein. Der Ausdruck "Bug" steht im Programmiererslang für einen Programmfehler. "Debugging" bedeutet demnach die Entfernung von Fehlern.

Dabei wäre es manchmal günstig zu wissen, was ein Programm an bestimmten Stellen denn genau macht. Sprich, man müßte sich während des Programmlaufs Variablen ansehen können u.ä. Genau dafür ist ein Debugger da. Er arbeitet das Programm im "Einzelschrittverfahren" (Trace-Modus) ab, so daß Sie nach jedem Befehl die Möglichkeit haben, Speicherstellen zu überprüfen oder das Programm (geringfügig) zu ändern.

1.4 Der Aufbau eines Computers

Zunächst müssen wir etwas über den Aufbau des Amiga erfahren. Speziell die Assemblersprache erfordert recht gute Kenntnisse des Systems, die man sich aber zum großen Teil auch während der Laufbahn als Programmierer aneignen wird (Übung macht ja bekanntlich den Meister).

1.4.1 Der Speicher - Platz für Programme

Der Speicher läßt sich als lange Reihe von Zellen ansehen (beim Amiga bis zu 10 Millionen). Er ist in zwei große Bereiche aufgeteilt: Das RAM (Random Access Memory), den Speicher mit wahlfreiem Zugriff, den man also lesen und beschreiben kann, und das ROM (Read Only Memory), das nur gelesen werden kann. Außerdem geht der Inhalt des RAM verloren, wenn Sie den Computer ausschalten, der des ROM aber bleibt permanent gespeichert. In letzterem liegen daher die Programmteile, die dafür sorgen, daß der Amiga beim Einschalten überhaupt etwas tun kann (das sog. Betriebssystem).

In jede dieser Zellen kann man eine Zahl zwischen 0 und 255 abspeichern. Wie wir gesehen haben, stellt ein Maschinenprogramm auch nur eine Folge von Zahlen dar, weshalb es sich problemlos im Speicher ablegen läßt. Jede Zelle erhält weiterhin eine Nummer, eine Adresse, die zum Zugriff auf sie benutzt wird.

1.4.2 Die CPU, das Herz des Computers

Die CPU (Central Processing Unit - Zentrale Steuereinheit), beim Amiga ein MC68000-Chip, ist gleichsam das Herz eines Computers. Sie ist es letztendlich, die unsere Programme ausführt. Sie kann rechnen, vergleichen, anhand der Vergleiche Entscheidungen fällen usw. Dabei bedient sie sich eines Verfahrens, das "Fetch and Execute" (Holen und Ausführen) genannt wird. Das bedeutet, die CPU holt sich einen Befehl aus dem Speicher, führt ihn aus, geht dann zum nächsten, holt ihn usw. Ein spezielles Register in der CPU, genannt PC (Program Counter - Programmzähler), enthält immer die Adresse, auf welche die CPU gerade zugreift. Nach jedem Befehl wird der PC erhöht. Damit ist auch klar, wie GOTO in Assembler realisiert wird: der PC wird mit der anzuspringenden Adresse geladen, und schon läuft das Programm dort weiter.

Die MC68000-CPU kennt zwei Betriebsarten: Den User- und den Supervisor-Modus. Im Supervisor-Modus (das bedeutet Überwacher-Modus) ist die Benutzung sämtlicher Befehle erlaubt, während im User-Modus einige Befehle, die bei unbedachter Anwendung schnell zum Systemabsturz führen können, verboten

sind. Außerdem benutzt die CPU getrennte Stackpointer (kommt später noch) für User- und Supervisor-Modus.

1.4.3 Der Bus - Verbindung zwischen CPU und Speicher

Nun kommt ein weiterer Bestandteil des Computersystems ins Spiel: der Bus. Es gibt einen Daten- und einen Adreßbus. Wenn die CPU ein Programm ausführen soll, muß sie Daten aus dem Speicher auslesen. Das tut sie, indem Sie die Adresse der gewünschten Speicherstelle auf den Adreßbus legt. Ein weiterer Baustein, die MMU (Memory Management Unit - Speicherverwaltungs-Einheit), sorgt aufgrund dieser Adresse dafür, daß der richtige RAM-Chip angesprochen wird. Der Inhalt der adressierten Speicherstelle gelangt dann über den Datenbus zurück zur CPU.

Analog verhält es sich, wenn die CPU in den Speicher schreibt. Sie legt die zu schreibende Zahl auf den Datenbus und die Adresse der Speicherstelle auf den Adreßbus, sagt der MMU, daß geschrieben werden soll, und schon läuft die Sache (meistens).

Es gibt dabei natürlich ein paar Einschränkungen. Greift z.B. die CPU auf eine Adresse zu, der gar keine Speicherstelle entspricht, oder will sie einen Befehl an einer ungeraden Adresse ausführen, wird bestimmt der Guru wieder mal zuschlagen.

1.5 Die verschiedenen Zahlensysteme

Das Zahlensystem, das wir für gewöhnlich benutzen (Zehnersystem), ist für den Gebrauch in einem Computer recht unpraktisch. Warum das so ist und welche Zahlensysteme stattdessen verwendet werden, erfahren Sie in diesem Abschnitt.

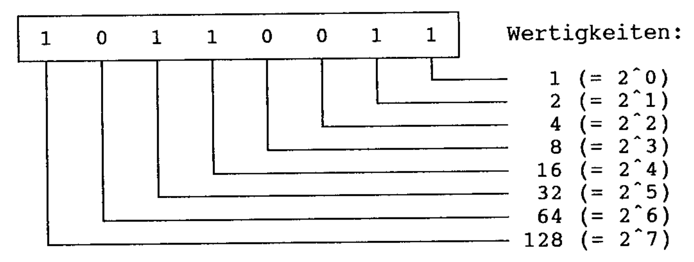
1.5.1 Bits und Bytes

Keine Sorge, dies ist keine Bier-Schleichwerbung. Als Bit bezeichnet man die kleinste von einem Computer darstellbare Informationseinheit. Ein Bit kann nur zwei Werte annehmen: 0 oder 1. Es stellt also quasi den Zustand eines Computer-Schaltkreiselements (Strom fließt oder Strom fließt nicht) dar. Acht solcher Bits werden zu einem Byte zusammengefaßt. Das Byte bildet dann die Grundlage des Speichersystems: Jede Speicherzelle ist genau ein Byte (oder acht Bit) groß. In einem Byte lassen sich also Werte zwischen 00000000 und 11111111 darstellen. Diesen Zahlen entsprechen die Zahlen 0 und 255 im Dezimalsystem.

1.5.2 Das Dualsystem

Die Bezeichnung "Zehnersystem" rührt von der Basis dieses Zahlensystems her: Jede Stelle einer Dezimalzahl hat eine um das zehnfache höhere Wertigkeit als die Stelle rechts daneben. Nun kann man im Prinzip jede beliebige Zahl als Basis für ein Zahlensystem annehmen. Im Dualsystem (oder auch "Binärsystem") ist dies die Zahl 2, d.h. jede Stelle hat gegenüber der Stelle rechts daneben die doppelte Wertigkeit. Vorkommen können in einer Dualzahl nur die Ziffern 0 und 1, genauso wie im Zehnersystem die Ziffern 0 bis 9 vorkommen.

Für eine Umrechnung von dual in dezimal betrachtet man am besten die Wertigkeiten der einzelen Dualstellen. Sie betragen nämlich $2^{\text{Binärstelle}}$, wobei die Stelle von 0 ab gezählt wird. Ein Beispiel:



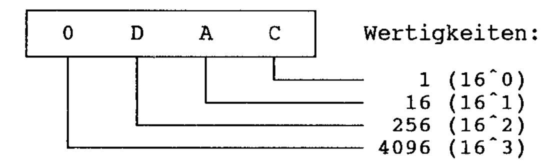
Die Wertigkeiten der Stellen mit Dual-1 zählt man zusammen: 128+32+16+2+1 = 179. Der Dualzahl 10110011 entspricht also dezimal 179.

Nun ist auch klar, warum ein Byte (8 Bit) Werte zwischen 0 und 255 annehmen kann: ein Byte kann höchstens die Dualzahl 11111111 enthalten (alle 8 Bits auf 1), was dezimal 128+64+32+16+8+4+2+1 = 255 ist.

Nun wäre es aber ziemlich unpraktisch, alle Zahlen, die man im beim Programmieren braucht, in dualer Schreibweise anzugeben, z.B. wäre die Zahl 3500 in dual 110110101100. Daher benutzt man im allgemeinen ein anderes Zahlensystem, das in recht engem Zusammenhang zu dem dualen steht:

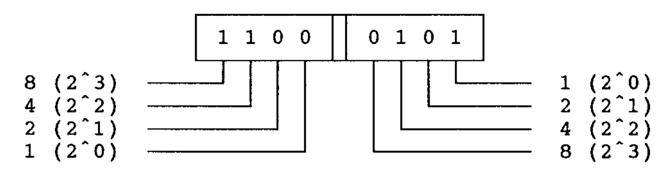
1.5.3 Das Hexadezimalsystem

"Hexadezimal" steht für die Zahl 16, also ist die Basis dieses Systems die 16. Das bedeutet, jede Ziffer hat eine um das 16-fache höhere Wertigkeit als die Ziffer rechts von ihr, und es kommen Ziffern von 0 bis 15 vor. Da es aber nur die Zahlzeichen 0 bis 9 gibt, benutzt man im Hexadezimalsystem die Buchstaben A bis F als Ersatz für 10 bis 15. Die Umrechnung erfolgt analog zum Dualsystem (die Stellenwertigkeit beträgt hier $16^{\text{Hexstelle}}$):



Man rechnet: 0*4096+13*256+10*16+12 = 3500 (D steht für 13, A für 10 usw.). Die Zahl 3500 läßt sich in hexadezimal also als 0DAC (oder einfach DAC) schreiben, während wir in dual 110110101100 schreiben mußten.

Der Zusammenhang zwischen hex und dual ist folgender: Jeweils vier Dualstellen stehen für eine Hex-Stelle. Beispiel: Die Dualzahl 11000101, also dezimal 197, teilt man in zwei Vierergruppen auf. Mit Wertigkeitstabelle sieht das dann so aus:



Getrennt ergeben die beiden Vierergruppen die Dezimalwerte 12 und 5. Für 12 schreibt man in hex C, somit heißt die Hex-Zahl C5. Fix umgerechnet ergibt das 12*16+5 = 197, das entspricht dem selben Wert wie die Dualzahl. Eine Zahl aus vier Dualstellen, also ein halbes Byte, nennt man übrigens "Nibble".

1.5.4 Das Oktalsystem

Ein weiteres Zahlensystem, das erwähnt werden soll, obwohl es fast nie zum Einsatz kommt, ist das Oktalsystem. Die Basis dieses Systems ist die 8, Aufbau und Umrechnung von Oktalzahlen sind analog zu den sonstigen Systemen.

1.5.5 Kennzeichnung der Zahlensysteme

In Assembler (und auch in vielen anderen Sprachen) hat es sich bei Benutzung von Zahlen eingebürgert, als Kennzeichnung des für die Zahl verwendeten Systems bestimmte Zeichen vor die Zahl zu setzen, und zwar:

Zahlensystem	Zeichen	Beispiel
Hexadezimal	$	$0DAC
Dual (Binär)	%	%10110011
Dezimal		179
Oktal	&	&263

Bild 1.1: Die Kennzeichen der Zahlensysteme

1.5.6 Datentypen: Bytes, Worte und Langworte

Wie schon gesagt, ist jede Speicherstelle ein Byte groß, kann also Zahlen von 0 bis 255 enthalten. Für die meisten Zwecke dürfte dieser Zahlenbereich wohl nicht ausreichen, daher gibt es noch zwei weitere "Größenordnungen": Zum einen das "Wort" (Word), das aus zwei Bytes besteht. Darin lassen sich dann schon Zahlen von 0 bis 65535 speichern. Die dritte Stufe ist das "Langwort" (Longword), dieses ist vier Bytes groß und kann Zahlen von 0 bis 4294967295 speichern (was wohl für so ziemlich alle Zwecke ausreichen dürfte). Diese drei Stufen werden "Datentypen" genannt.

Vielleicht kennen Sie den Begriff des Datentyps schon von der Programmiersprache C her. Wenn Sie jetzt geschockt sind, weil Sie glauben, daß Sie (wie in C) mit einer Riesenmenge an Datentypen konfrontiert werden, können Sie beruhigt sein: In Assembler gibt es nur die drei Typen Byte, Wort und Langwort.

Wenn ein Wort oder Langwort im Speicher abgelegt werden soll, werden dafür einfach 2 bzw. 4 Bytes benutzt. Das hat zur Folge, daß Worte und Langworte, genau wie Assemblerbefehle, nur an geraden Adressen beginnen dürfen, ansonsten reist der Amiga wieder mal nach Indien (Guru).

Die Unterscheidung dieser drei Datengrößen ist in Assembler sehr wichtig. Bei fast allen Befehlen, die mit Daten umgehen, muß angegeben werden, auf welche Größe sie sich beziehen. Für Adressen müssen beim Amiga (fast) immer Langworte verwendet werden.

1.5.7 Der ASCII-Code - jedem Zeichen eine Zahl

Das Thema ASCII-Code paßt eigentlich nicht so ganz in das Kapitel "Zahlensysteme", denn es handelt sich dabei um kein solches. Aber der ASCII-Code ist eine grundlegende Sache und einem Zahlensystem recht ähnlich, weshalb wir ihn hier besprechen wollen.

Die Abkürzung ASCII steht für "American Standard Code for Information Interchange", also "Amerikanischer Standard-Code für Informationsaustausch". Es handelt sich dabei um eine einheitliche Darstellungsmethode von Zeichen und Steuerco-

des, die auf jedem Computersystem gleich ist (bzw. sein sollte). Die ASCII-Codierung hat nichts mit Chiffrierung von Texten zu tun. Vielmehr wird jedem auf dem Bildschirm darstellbaren oder ausführbaren Zeichen eine Zahl zugeordnet. Diese Zahl liegt zwischen 0 und 255, womit 256 verschiedene Codes möglich sind.

Die ersten 32 Codes (also 0 bis 31) sind sog. "Steuercodes", d.h. sie stellen kein druckbares Zeichen wie einen Buchstaben oder eine Zahl dar, sondern sie lösen, wenn sie ausgegeben werden, bestimmte Sonderfunktionen aus. Der Code 10 z.B. läßt den Cursor in die nächste Zeile springen (wie ein Druck auf die Return-Taste), 12 löscht den Bildschirm, 8 führt einen Rückschritt aus (wie die Backspace-Taste) und 9 steht für die Tabulator-Taste.

Ab Code Nr. 32 beginnen die druckbaren Zeichen. 32 ist die Leertaste, 33 das '!'-Zeichen, 34 das Anführungszeichen, 35 das Doppelkreuz ('#') usw. Von 48 bis 58 liegen die Zahlen 0 bis 9. Von 65 bis 90 die Großbuchstaben und von 97 bis 122 die Kleinbuchstaben. Ab 127 beginnen die Sonderzeichen, die allerdings von System zu System verschieden sein können (mit dem "Standard-Code" war's wohl doch nicht so ganz).

Diese Beispiele sollen Ihnen nur klarmachen, wie die ASCII-Codierung funktioniert. Im Anhang finden Sie eine komplette Tabelle aller ASCII-Zeichen. Wenn Sie von nun an den Begriff "ASCII" lesen, wissen Sie, was gemeint ist.

Kapitel 2
Programmier-Grundlagen

Der Registersatz

Das Statusregister

Die Typenangabe

Sinn und Zweck des Stacks

Einführung in die Adressierung

Die Adressierungsarten komplett

Wichtige Assembler-Befehle

Darstellung negativer Zahlen

2.1 Der Registersatz des MC68000

Bis jetzt haben wir nur davon gesprochen, daß es einen Speicher gibt, in dem Daten abgelegt werden können. Die Amiga-CPU stellt uns aber neben dem großen Hauptspeicher noch einen kleineren, speziellen "Arbeitsspeicher" zur Verfügung. Dieser Speicher liegt direkt auf dem CPU-Chip, weshalb es keine Zeitverzögerung durch Benutzung des Bus-Systems gibt. Er ist, im Gegensatz zum Hauptspeicher, in Langworte (4 Bytes) aufgeteilt. Jedes dieser Langworte nennt man ein "Register".

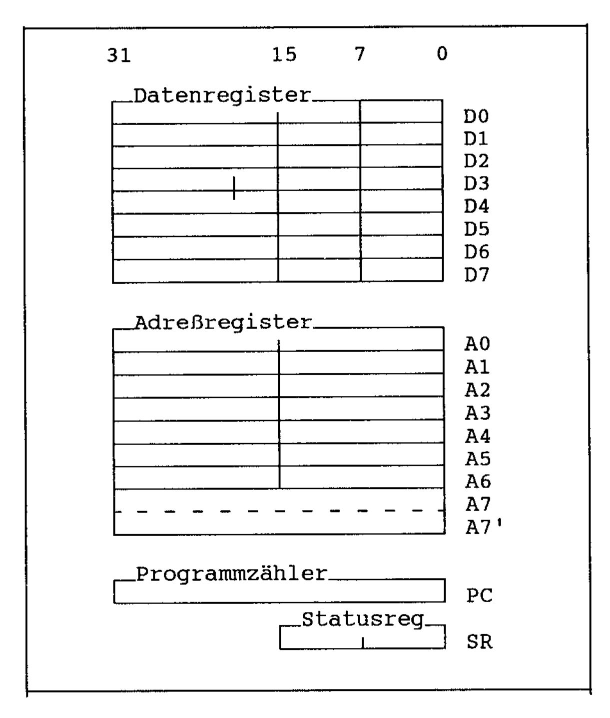


Bild 2.1: Der Registersatz des MC68000

Die Zahlen 0-31 über den Kästen stellen die Bitnummern dar. Die senkrechten Unterteilungen sollen verdeutlichen, daß man diese Register als Byte, Wort oder Langwort ansprechen kann.

Speicherstellen werden, wie wir wissen, über ihre Adressen angesprochen. Bei den Registern verwendet man zur "Anrede" stattdessen ihre Namen (d0, d1, usw.).

Die CPU verfügt, wie Sie aus dem Bild 2.1 ersehen können, über 8 Datenregister (die als Byte, Wort oder Langwort angesprochen werden können), 8 Adreßregister (nur Wort oder Langwort), den PC (immer Langwort) und das Statusregister.

Das Adreßregister A7, der Stackpointer (siehe nächster Abschnitt), ist quasi "zweigeteilt". A7 ist der Stackpointer für den User-Modus und A7' der für den Supervisor-Modus. Das heißt aber nicht, daß Sie nach Belieben A7 und A7' in Ihren Programmen benutzen können. Die Aufteilung soll nur verdeutlichen, daß im User- und Supervisor-Modus getrennte Stackpointer benutzt werden.

2.1.1 Das Statusregister

Dieses ein Wort breite Register ist aufgeteilt in das User-Byte und das System-Byte. Die Bits dieses Registers werden auch "Flags" genannt. Ein Flag ist eine Art Schalter, der nur zwei Zustände annehmen kann (also wie ein Bit). Man sagt auch, ein Flag ist gesetzt (steht auf 1) oder gelöscht (nicht gesetzt, steht auf 0). Die Bedeutung der Flags ist folgende:

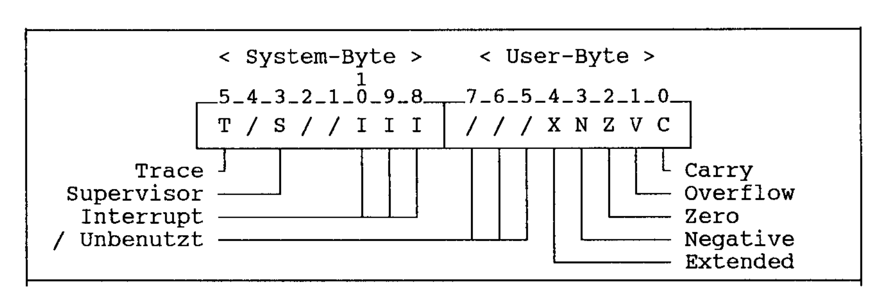


Bild 2.2: Das Statusregister und die Flags

Das System-Byte enthält die drei System-Flags T, S und I. Diese können wir nicht so ohne weiteres ändern (das geht nur im Supervisor-Modus). Interessanter ist für uns das User-Byte, genannt "Condition Code Register" (CCR, bedeutet "Bedingungscode-Register"). Fast alle Assembler-Befehle beeinflussen nämlich die Flags des CCR, oder besser, die Flags werden gemäß dem Ergebnis des Befehls gesetzt (oder gelöscht). Beispiel: Wenn eine Operation den Wert 0 liefert (z.B. eine Subtraktion, oder auch ein MOVE-Befehl, der den Wert 0 bewegt), wird das Z-Flag (Z steht für Zero, also Null) gesetzt. Lieferte sie einen Wert ungleich 0, wird das Z-Flag gelöscht.

Analog arbeiten auch die übrigen Flags. Das C-Flag (C für Carry, das bedeutet Übertrag) wird gesetzt, wenn eine Rechen-Operation einen Bereichsüber- oder -unterlauf

verursachte, und gelöscht, falls dies nicht passierte. Das N-Flag (Negative) arbeitet im Zusammenhang mit positiven und negativen Rechenergebnissen, das V-Flag (Overflow) wird bei Überschreitung des Wertebereichs gesetzt, und das X-Flag (Extended, Erweiterung) wird von bestimmten Befehlen als "Bit-Zwischenspeicher" benutzt.

Die Flags spielen eine große Rolle bei Vergleichen und bedingten Sprüngen, daher werden sie im Abschnitt über die Programmsteuer-Befehle (2.5.4) näher behandelt.

2.1.2 Typenangabe bei Assembler-Befehlen

Da der Amiga wissen muß, auf welchen Datentyp sich Ihre Befehle beziehen, ist es wichtig, an den Befehl die Typenangabe anzuhängen. Will man z.B. den Inhalt des Datenregisters D0 an die Speicherstelle 1000 kopieren, schreibt man:

```
move.b  d0,1000      oder
move.w  d0,1000      oder
move.l  d0,1000
```

Der MOVE-Befehl mit .b (Byte) kopiert nur die Bits 0-7, also das unterste Byte des Registers, .w (Wort) kopiert die Bits 0-15, .l (Langwort) das ganze Register. Wie schon erwähnt, belegt ein Wort 2 Bytes im Speicher und ein Langwort 4. Wenn in d3 also das hex-Langwort $12345678 stünde, würde im Falle .b der Wert $78 in die Speicherstelle 1000 geschrieben, bei .w $56 in 1000 und $78 in 1001, bei .l $12 in 1000, $34 in 1001, $56 in 1002 und $78 in 1003. Es ist wichtig, daß Sie beim Moven (und bei ähnlichen Operationen) immer den richtigen Datentyp angeben, sonst könnten Sie schnell wichtige Daten im Speicher überschreiben.

Die Unterteilung in Daten- und Adreßregister dürfen Sie nicht allzu eng sehen. Sie können durchaus auch Daten in Adreßregistern speichern und umgekehrt. Es gibt aber bestimmte Befehle (z.B. Multiplikation), die nur auf Datenregister angewendet werden dürfen, und bestimmte andere Befehle (oder Befehlsschreibweisen), die nur mit Adreßregistern zulässig sind. Außerdem sollten Sie beachten, daß der Datentyp Byte in Adreßregistern nicht möglich ist. Die Einhaltung dieser Regeln überwacht aber normalerweise sowieso der Assembler (das Übersetzungsprogramm).

2.2 Sinn und Zweck des Stacks

Es kommt sehr häufig vor, daß der Programmfluß eines Assemblerprogramms unterbrochen, ein anderer Programmteil ausgeführt und anschließend wieder zur Unterbrechungsstelle zurückgekehrt wird. Dies bezeichnetm man als "Ausführung von Unterprogrammen" oder "Sub-Routinen". Ein Anwendungsbeispiel

wäre eine Routine, die einen Text zentriert auf dem Bildschirm ausgibt. Zu diesem Zweck muß man aus der Länge des Textes seine Horizontal-Startposition in der Zeile ausrechnen. Anstatt nun an jeder benötigten Stelle im Programm die komplette Berechnungsroutine einzusetzen, schreibt man diese nur einmal, eben als Unter-Programm, und läßt sie von den entsprechenden Stellen im Hauptprogramm aufrufen. Die Parameter (Länge des Textes, Zeiger auf den Text selber) könnte man in Registern ablegen und sie vom Unterprogramm auswerten lassen.

Damit der Computer aber vom Unterprogramm wieder ins Hauptprogramm zurückfindet, muß er sich natürlich merken, an welcher Stelle es unterbrochen wurde. Zu diesem Zweck dient der Stack (zu Deutsch Stapel). Eigentlich ist der Stack nur ein ganz gewöhnliches Stück des Speichers, erst durch seine Benutzung als Stack wird er zu etwas Besonderem.

Die Bezeichnung Stapel kommt dabei nicht von ungefähr: Man kann auf dem Stack, wie auf einem Papierstapel, Daten (oder auch Adressen) ablegen und sie später wieder herunternehmen. Das geht allerdings immer nur "von oben", man kann also nur das herunternehmen, was man als letztes draufgelegt hat. In der Fachsprache nennt sich das LIFO-Prinzip (LIFO = Last In - First Out). Wie wird nun der Stack beim Amiga realisiert?

Das Adreßregister a7 spielt in diesem Zusammenhang eine wichtige Rolle: Es ist der "Stackpointer", also der Stapelzeiger. Über ihn kann man den Platz des Stapels, der als nächstes belegt werden soll, in Erfahrung bringen, denn er zeigt immer auf den Platz, der als letztes belegt worden ist. Das klingt vielleicht ein bißchen kompliziert, leuchtet aber ein, wenn man weiß, was beim "Ablegen auf dem Stack" eigentlich passiert. Sagen wir zum Beispiel "Lege das Datenregister d0 auf den Stack", heißt das für den Computer:

1. Erniedrige den Stackpointer
2. Schreibe d0 auf die Adresse, auf die der Stackpointer jetzt zeigt.

Wenn nun weitere Daten auf den Stack sollen, wird wieder der Stackpointer erniedrigt und anschließend werden die Daten geschrieben. Auf diese Weise kann man beliebig viele Daten auf den Stack schreiben (jedenfalls solange der Speicherplatz reicht) ohne vorhergehende Daten zu überschreiben. Wenn wir nun unsere Daten wieder vom Stack holen möchte (z.B. nach d0), passiert folgendes:

1. Schreibe den Inhalt der Speicherstelle, auf die der Stackpointer zeigt, nach d0.
2. Erhöhe den Stackpointer

Damit haben wir unsere Daten wieder, und der Platz, den sie vorher auf dem Stack einnahmen, ist wieder "frei" (durch die Erhöhung des Stackpointers).

Da der Stackpointer vor dem Ablegen erniedrigt und nach dem Herunternehmen erhöht wird, zeigt er immer auf das, was als letztes auf den Stack gelegt wurde. Außerdem wächst er quasi "von oben nach unten", d.h. von den höheren zu den niedrigeren Speicherstellen. Das ist wichtig zu wissen, wenn wir uns den Inhalt des Stacks anschauen wollen, ohne etwas herunterzunehmen (wir also den Papierstapel in der Mitte durchwühlen wollen). Wir könnten z.B. sagen: Schreibe das, was an Stackpointerposition+2 steht, nach d0. Damit haben wir uns den drittletzten Wert auf dem Stack geholt, ohne ihn vom Stack zu löschen. Holen UND Löschen ist nur von oben möglich.

Unterprogrammaufrufe laufen ähnlich wie die Datenspeicherung ab. Der Befehl "Springe Unterprogramm an Adresse 1000 an" bewirkt folgendes:

1. Erniedrige den Stackpointer
2. Schreibe die Adresse des Befehls nach dem Sprungbefehl in den Speicher (dahin, wo der Stackpointer jetzt hinzeigt).
3. Lade den PC mit der Adresse 1000 (das Programm läuft dann dort weiter).

Beim Ende des Unterprogramms (Rücksprung) passiert das:

1. Hole den Inhalt der Speicherstelle, auf die der Stackpointer zeigt.
2. Erhöhe den Stackpointer.
3. Springe zu der geholten Adresse.

Auf diese Weise lassen sich Unterprogrammaufrufe auch problemlos schachteln, dann wird der Stack bei jedem Aufruf um eins größer und bei jedem Rücksprung wieder um eins kleiner.

Außer zur Speicherung von Rücksprungadressen dient der Stack oft auch zur Parameterübergabe, insbesondere beim Einbau von Assemblerroutinen in Hochsprachen-Programme. Das könnte so aussehen: Das Hauptprogramm legt zuerst die Return-Adresse auf den Stack und anschließend die Parameter. Dann wird verzweigt. Das Unterprogramm holt sich die Parameter vom Stack, läßt die Return-Adresse aber drauf. Beim Rücksprung ist später nur noch die Return-Adresse auf dem Stack, die automatisch richtig genutzt wird.

Wie gesagt ist beim Amiga das Adreßregister a7 der Stackpointer. Daher wird es auch oft sp genannt. Sie dürfen es in Ihren Programmen keinesfalls einfach so verändern (zur Speicherung von Adressen o.Ä.).

2.3 Einführung in die Adressierungsarten

Ein wichtiges Kriterium für die Leistungsfähigkeit einer CPU ist (neben dem Register- und Befehlssatz) die Anzahl verschiedener Adressierungsarten. In dieser Hinsicht gehört der MC68000 zu den besten CPUs, seine vielfältigen Möglichkeiten auf diesem Gebiet lassen kaum Wünsche offen. Aber zunächst wird Sie wohl interessieren, was Adressierungsarten überhaupt sind.

Wenn wir ein Adreßregister ansprechen wollen, können wir schreiben:

```
move.l   a0,a1
```

Damit wird der Inhalt von a0 direkt nach a1 kopiert. Das wäre schon die erste Adressierungsart: Register direkt. Das geht natürlich genausogut mit Datenregistern:

```
move.l   d0,d1
```

Dieser Befehl kopiert d0 nach d1. Wenn wir jetzt aber schreiben

```
move.l   (a0),(a1)
```

sind nicht die Register direkt gemeint, sondern die Werte, die in den Registern stehen, werden als Adressen (Speicherstellen) angesehen. Auf die Inhalte dieser Speicherstellen bezieht sich dann der Befehl. Steht z.B. in a0 eine 1000 und in a1 eine 2000, wird durch den MOVE-Befehl der Inhalt von Adresse 1000 in die Adresse 2000 kopiert. Diese Adressierungsart nennt sich "Adreßregister indirekt", abgekürzt ARI. Sie ist, wie der Name schon sagt, Adreßregistern vorbehalten.

Gehen wir noch einen Schritt weiter: Angenommen, wir haben in a0 die Anfangsadresse einer Tabelle stehen, die wir im Speicher angelegt haben, und möchten nun Byte für Byte der Tabelle bearbeiten. Man könnte nun mittels

```
move.b   (a0),d0
```

das erste Tabellen-Byte zur Bearbeitung nach d0 kopieren und dann a0 mit einem weiteren Befehl um eins erhöhen. Schöner ist es aber, diese Erhöhung automatisch durchführen zu lassen:

```
move.b   (a0)+,d0
```

Damit wird die Speicherstelle, auf die a0 zeigt, nach d0 kopiert, und a0 dann automatisch um 1 erhöht. Praktisch, nicht? Aber es kommt noch besser: Ein Wort belegt bekanntlich 2 Byte, ein Langwort 4. Bei Abarbeitung einer Wort-Ta-

belle müßte der Zeiger also jedesmal um 2 erhöht werden. Und das wird er auch:

```
move.w   (a0)+,d0
```

Dieser Befehl holt das Wort nach d0 und erhöht a0 dann um 2. Ebenso würde ein move.l das Register um 4 erhöhen. Diese Adressierungsart nennt sich "ARI mit Postinkrement".

Eine andere Variante des ARI ist "ARI mit Predekrement":

```
move.b   d0,-(a0)
```

Hier wird a0 also zuerst erniedrigt und dann d0 in die betreffende Speicherstelle kopiert. Wenn Sie nun an den Abschnitt über den Stack denken, fällt vielleicht langsam der Groschen: Mit dieser Adressierungsart ist es kein Problem, Daten auf den Stack zu legen und wieder herunterzunehmen. Beim Ablegen sollte der Stackpointer erniedrigt werden und dann das Abzulegende dahin geschrieben werden, wohin der Stackpointer zeigt. Das tut genau diese Adressierungsart:

```
move.l   d0,-(sp)
```

würde demnach d0 auf den Stack bringen (wo es vier Bytes belegt), und

```
move.l   (sp)+,d0
```

würde es zurückholen (mit "Löschung" vom Stack). Anstatt sp hätten wir natürlich auch a7 schreiben können.

2.4 Adressierungsarten komplett

Als nächstes folgt eine Auflistung aller 12 Adressierungsarten des MC68000, natürlich mit Erklärung.

Adressierungsart	Abkürzung	Beispiel
Datenregister direkt	Dn	move.l d0,d1
Adreßregister direkt	An	move.l a0,a1
Adreßregister indirekt (ARI)	(An)	move.l (a0),(a1)
ARI mit Postinkrement	(An)+	move.l (a7)+,d0
ARI mit Predekrement	-(An)	move.l d0,-(a7)
ARI mit Adreßdistanz	d16(An)	move.l 2(a7),d0
wie vor plus Index	d8(An,Rn)	move.l 0(a0,d0),d1
Absolut kurz	$xxxx	move.l $1000,d0
Absolut lang	$xxxxxxxx	move.l $fc0004,d0
Konstante	#x	move.l #1,d0
PC-Relativ + Adreßdistanz	d16(PC)	move.l 20(pc),d0
wie vor plus Index	d8(PC,Rn)	move.l 0(pc,d0),d1

Bild 2.3: Die Adressierungsarten des MC68000

2.4.1 Register direkt

Ein Daten- oder Adreßregister wird direkt angesprochen.

```
move.l  d0,d1
move.l  a0,a1
```

2.4.2 Adreßregister indirekt (ARI)

Der Inhalt eines Adreßregisters wird als Speicherstelle angesehen, auf die dann der Befehl ausgeführt wird.

```
move.l  d0,(a0)
```

Der Inhalt von d0 wird an die Speicherstelle kopiert, die in a0 steht.

2.4.3 ARI mit Postinkrement

Siehe ARI, nur wird hier das Adreßregister nach dem Zugriff erhöht, und zwar je nach Datentyp um 1, 2 oder 4.

```
move.b  d0,(a0)+        ; Erhöhung um 1
move.w  d0,(a0)+        ; Erhöhung um 2
move.l  d0,(a0)+        ; Erhöhung um 4
```

2.4.4 ARI mit Predekrement

Wie vorher, aber hier wird vor dem Zugriff erniedrigt.

```
move.b  d0,-(a0)        ; Erniedrigung um 1
move.w  d0,-(a0)        ; Erniedrigung um 2
move.l  d0,-(a0)        ; Erniedrigung um 4
```

2.4.5 ARI mit Adreßdistanz

Hier wird zum Inhalt des Adreßregisters noch eine konstante Zahl, die Adreßdistanz (oft auch "Offset" genannt), hinzuaddiert (bzw. abgezogen). Diese Zahl darf zwischen +32767 und -32768 liegen. Die Summe aus Distanz und Registerinhalt ist die Adresse, auf die sich der Befehl bezieht.

```
move.l  d0,-20(a0)
```

Steht in a0 eine 100, wird d0 also nach 80 kopiert. Dies ist eine Adressierungsart, die beim Umgang mit dem Amiga-Betriebssystem recht häufig benutzt wird.

2.4.6 ARI mit Adreßdistanz und Index

Neben der Adreßdistanz (der konstanten Zahl) wird nun noch ein weiteres, frei wählbares Register zum Inhalt des Adreßregisters hinzuaddiert. Die Distanz darf jetzt nur noch im Bereich von +127 bis -128 liegen. Der Datentyp des zweiten Registers ist (unabhängig vom Typ des Adreßregisters) frei wählbar (bei einem zweiten Adreßregister aber natürlich nur Wort oder Langwort).

```
move.l   d0,10(a0,d1)
```

Steht in a0 eine 1000 und in d1 eine 20, wird sich der Befehl auf die Adresse 1030 beziehen. Denkbar wäre auch:

```
move.l   d0,10(a0,d1.w)
```

Das kann man schreiben, wenn das Register d1 nicht komplett (als Langwort), sondern nur als Wort in die Rechnung eingehen soll (wenn es also nur Werte zwischen +32767 und -32768 speichern soll).

Diese Adressierungsart ist sehr nützlich beim Abarbeiten von Tabellen. Im Adreßregister steht dann die Anfangs-(Basis-) Adresse der Tabelle und im zweiten Register die Platznummer. Oft wird die Adreßdistanz dabei nicht gebraucht, weshalb man folgendes schreibt:

```
move.l   d0,0(a0,d1.w)
```

Die Adreßdistanz ist 0, also gehen nur Basisadresse und Platznummer (Index) in die Rechnung ein.

2.4.7 Absolute Adressierung

Zur Abwechslung eine einfache Adressierungsart: Hier wird eine Speicherstelle direkt angesprochen:

```
move.l   1000,2000
```

Der Inhalt von Adresse 1000 wird nach 2000 kopiert. Beachten Sie, daß sich Wort- und Langwort-Befehle nur auf gerade Adressen beziehen dürfen (Guru-Gefahr)!

Bei der absoluten Adressierung wird zwischen einer Kurz- und einer Lang-Version unterschieden. Kurz bedeutet, daß die Adreßangabe nur ein Wort breit ist, d.h. die Adresse darf nur zwischen 0 und 65535 liegen. Bei Lang ist die Adresse ein Langwort, kann also im kompletten Adreßbereich liegen. Die Kurz-Version der absoluten Adressierung hat den Vorteil, daß zur Speicherung der Adresse nur zwei Bytes benötigt werden (im Gegensatz zu 4 Bytes bei Lang).

Normalerweise wählt der Assembler automatisch die Kurz-Version, wenn dies möglich ist. Sie können darüber aber auch

selbst bestimmen, indem Sie .w oder .l an die Adresse anhängen:

```
move.l    2000.w,100000.l
```

Die Adresse 2000 liegt im Wort-Bereich, daher kann sie als Wort-Adresse assembliert werden. Die 100000 liegt aber außerhalb dieses Bereiches, muß also Lang sein.

2.4.8 Konstanten-Adressierung

Die wohl einfachste Adressierungsart. Um eine Zahl als Konstante anzusprechen, schreiben Sie einfach ein '#' vor die Zahl.

```
move.l    #1,d0
```

Schreibt die Zahl 1 ins Datenregister d0. Achten Sie darauf, daß Sie nicht die Adressierungsarten "Absolute Adresse" und "Konstante" verwechseln!

2.4.9 PC-relative Adressierung

Um den Sinn dieser recht interessanten Adressierungsart zu verstehen, müssen Sie vorab etwas wissen: Ein Programm für den Amiga, der ja bekanntlich ein Multitasking-Computer ist, darf nicht an eine bestimmte Position im Speicher gebunden sein, denn dort könnte ja schon ein anderes Programm stehen. Das Programm muß "lageunabhängig" (position independent) sein, darf also keine festen Adressen enthalten. Es gibt nun mehrere Möglichkeiten, das zu erreichen.

Die erste Möglichkeit wird (fast) immer, für Sie unsichtbar, vom Assembler benutzt: An das fertige Programm wird eine Tabelle mit allen verwendeten absoluten Adressen angefügt. Das Betriebssystem sorgt nun dafür, daß beim Laden des Programms an eine bestimmte Speicherposition die absoluten Adressen im Programm anhand der Tabelle umgerechnet werden.

Wenn Sie aber selbst für die Lageunabhängigkeit sorgen wollen (oder manchmal müssen), können Sie die Adressierungsart PC-relativ verwenden. Dabei wird die Adresse aus dem aktuellen PC-Stand plus (oder minus) einer Adreßdistanz berechnet. Dann ist es egal, an welcher Stelle das Programm im Speicher steht, da sich der Abstand adressierender Befehl - adressierte Speicherstelle ja nicht ändert. Das gilt natürlich nur für Adressen, die im Bereich des Programms liegen, weshalb PC-relativ nur innerhalb des Programm-Adreßbereichs benutzt werden darf. Die Adreßdistanz kann, wie bei ARI ohne Index, zwischen +32767 und -32768 liegen.

```
move.l    20(pc),d0
```

Dieser Befehl holt den Inhalt von pc+20 nach d0.

2.4.10 PC-relative Adressierung mit Index

Die PC-relative Adressierungsart gibt's auch noch mit Index. Es gelten die selben Regeln wie für "ARI mit Adreßdistanz und Index", nur ist hier der aktuelle PC-Stand die Basisadresse.

```
move.l   20(pc,d1.w),d0
```

Holt den Inhalt der Speicherstelle "PC-Stand plus 20 plus Inhalt von d1 als Wort" nach d0.

2.5 Die Assembler-Befehle

Der einzige Assembler-Befehl, den wir bisher benutzt haben, ist der MOVE-Befehl. Sicher brennen Sie schon darauf, zu erfahren, was der MC68000 noch so alles zu bieten hat (und das ist wirklich eine ganze Menge). In diesem Abschnitt werden daher die wichtigsten Befehle vorgestellt. Manche Befehle erfordern allerdings recht umfangreiche Erklärungen, daher werden diese erst in späteren Kapiteln ausführlich beschrieben.

Man könnte zwar auch so vorgehen, neue Kommandos erst dann zu erklären, wenn sie zum ersten mal verwendet werden. Wir halten es aber für sinnvoller, Ihnen zuerst einen Überblick zu geben, damit sie nicht wie der Ochs vorm Berg stehen, wenn ein neuer Befehl in einem Listing auftritt.

2.5.1 Typen von Assembler-Befehlen

Grob gesehen gibt es drei Typen von Assembler-Befehlen: Befehle ohne Operanden, mit einem oder mit zwei Operanden. Ein Beispiel für einen Befehl ohne Operanden ist:

```
rts                      ; Return from Subroutine
```

Dieser Befehl verläßt ein Unterprogramm. Um ihn ausführen zu können, braucht die CPU nichts weiter zu wissen (die Rücksprungadresse liegt ja auf dem Stack). Bei dem Befehl

```
clr.l    d0              ; Clear = Lösche
```

dagegen muß der Prozessor wissen, was er denn löschen soll, mit anderen Worten, er braucht einen Operanden (ein Objekt, auf das sich der Befehl bezieht). In diesem Fall ist d0 der Operand. Bei einem Befehl wie

```
move.l   d0,d1          ; Quelle und Ziel benötigt
```

werden sogar zwei Operanden gebraucht - die CPU muß ja wissen, von wo nach wo bewegt werden soll.

Im Zusammenhang mit den Adressierungsarten, die wir vorhin kennengelernt haben, ist es wichtig zu wissen, daß es für jeden Befehl bestimmte Vorschriften gibt, welche Adressierungsarten jeweils für Quell- und (ggf.) Zieloperand zulässig sind. Zu diesem Thema gibt es eine Tabelle im Anhang, aber auch der Assembler wacht in der Regel über die Einhaltung der Adressierungsregeln.

In den nun folgenden tabellarischen Auflistungen der Befehle werden folgende Abkürzungen benutzt:

Dn - beliebiges Datenregister
An - beliebiges Adreßregister
Rn - beliebiges Daten- oder Adreßregister
#k - Konstante
d - Adreßdistanz
ea - Daten-, Adreßregister oder Speicherstelle

"Ea" kann für ein Datenregister, ein Adreßregister oder eine "Effektive Adresse" stehen. Letzteres bedeutet einfach eine Adresse im Speicher. Diese kann durch absolute Adressierung, ARI (Adresse steht in einem Adreßregister), ARI mit Predekrement, ARI mit Offset (Adresse berechnet sich aus Register-Inhalt plus Konstante) usw. entstanden sein.

Es folgt jeweils zuerst eine kurze Einführung der generellen Bedeutung einer Befehlsgruppe, dann eine Tabelle mit den wichtigsten Befehlen einer Gruppe mit Kommentaren und, wenn nötig, noch einige Anmerkungen.

2.5.2 Transfer-Befehle

Sie dienen dazu, Daten "von irgendwo nach irgendwo anders" zu transportieren. Tatsächlich stehen bei den Transfer-Befehlen so ziemlich alle Möglichkeiten (Wahl der Adressierungsart, der Quelle und des Ziels) offen. Diese Befehlsart wird man wohl am häufigsten in einem Assembler-Listing finden.

Wichtig zu wissen ist übrigens, daß die MOVE-Befehle eigentlich COPY heißen müßten, da sie streng genommen Kopier-Befehle sind. Die Quelle wird lediglich ins Ziel kopiert, bleibt selbst aber unverändert.

Befehl	Bedeutung	Beispiele
CLR ea	Lösche	clr d0 clr 1000 clr (a0)
- Nicht für Adreßregister direkt		
EXG Rn,Rn	Vertausche Register	exg d0,d1 exg d0,a0 exg a0,a1
- Nur für Register erlaubt		
LEA ea,An	Lade eff. Adresse in An	lea 10(a0),a1 lea 20(pc),a1
- Rechnet eff. Adresse aus und schreibt sie in An.		
MOVE ea,ea	Kopiere Daten	move d0,d1 move (a0)+,1000 move #5,-(a1)
- Als Ziel ist Adreßregister direkt nicht erlaubt		
MOVEA ea,An	Kopiere Adresse	movea 1000,a0 movea #10,a1 movea 10(a0),a1
- Ziel darf hier nur Adreßregister direkt sein		
MOVEQ #k,Dn - k darf nur zwischen +127 und -128 liegen	Lade Dn "quick"	moveq #1,d0
MOVEM RL,ea MOVEM ea,RL - Siehe Anmerkungen	Kopiere Reg.liste	movem d0/a0,-(sp) movem (sp)+,d0/a0
SWAP Dn - Vertauscht Bits 0-15 mit Bits 16-31	Vertausche Worte von Dn	swap d0

Normalerweise brauchen Sie nicht zwischen MOVE und MOVEA zu unterscheiden. Der Assembler nimmt automatisch den richtigen Befehl, je nachdem, ob Sie als Ziel ein Adreßregister oder etwas anderes angeben.

Der MOVEQ-Befehl läßt als Ziel nur Datenregister zu. Wenn Sie also ein solches mit einer Zahl zwischen +127 und -128 belegen wollen, sollten Sie MOVEQ anstatt MOVE benutzen, da dieser schneller ausgeführt wird und weniger Speicher braucht.

SWAP wirkt immer auf ein Datenregister als Langwort. Das obere und untere Wort des Registers werden vertauscht.

Der MOVEM-Befehl eignet sich sehr gut zum Sichern von Registern auf dem Stack. Man kann ihn sich als Zusammenfassung mehrerer MOVE-Befehle denken: Anstatt

```
move.l   d0,-(sp)
move.l   d1,-(sp)
move.l   d2,-(sp)
move.l   d5,-(sp)
```

können wir auch schreiben:

```
movem.l  d0-d2/d5,-(sp)
```

Es sind beliebige Kombinationen aus Daten- und Adreßregistern erlaubt, und auch '-' (Bereiche) und '/' (einzelne Register) können beliebig kombiniert werden. Beispiele:

```
movem.l  d0-d7/a0-a6,-(sp)
```

sichert aller Register auf dem Stack (außer a7, denn dieses ist ja der Stackpointer selber). Noch ein Beispiel:

```
movem.l  d0-d2/a0/a2/a4-a6,-(sp)
```

Der LEA-Befehl bestimmt lediglich eine Adresse und schreibt sie in ein Adreßregister. Die Adresse kann sich je nach Adressierungsart aus absoluter Adresse, Adreßregisterinhalt, Adreßregisterinhalt plus Adreßdistanz usw. zusammensetzen.

2.5.3 Rechen-Befehle

Sie umfassen die vier Grundrechenarten und das Negieren (Vorzeichenwechsel). Als "Rechen-Richtung" gilt allgemein: Zieloperand verknüpft mit Quelloperand, Ergebnis in den Zieloperanden. Beispiel:

```
sub.l    d0,d1
```

zieht d0 von d1 ab (also d1 minus d0) und speichert das Ergebnis in d1.

Befehl	Bedeutung	Beispiele
ADD ea,ea	Addiere - Als Ziel ist Adreßregister nicht erlaubt	add #1,d0 add d0,1000
ADDA ea,An	Addiere Adresse - Ziel darf hier nur Adreßregister sein	adda #1,a0 adda d0,a0 adda 1000,a0
ADDQ #k,ea	Addiere Konstante "quick" - k darf nur zwischen 0 und 8 liegen	addq #1,d0 addq #1,1000

SUB ea,ea	Subtrahiere	sub #1,d0 sub d0,1000
- Als Ziel ist Adreßregister nicht erlaubt		
SUBA ea,An	Subtrahiere Adresse	suba #1,a0 suba d0,a0 suba 1000,a0
- Ziel darf hier nur Adreßregister sein		
SUBQ #k,ea	Subtrahiere Konstante "quick"	subq #1,d0 subq #1,1000
- k darf nur zwischen 0 und 8 liegen		
DIVS ea,Dn	Dividiere Wort mit Vorzeichen	divs #5,d0 divs (a0),d0
- Ergebnis kommt ins untere Wort von Dn, der Rest ins obere		
DIVU ea,Dn	Dividiere Wort ohne Vorzeichen	divu #5,d0 divu (a0),d0
- Ergebnis kommt ins untere Wort von Dn, der Rest ins obere		
MULS ea,Dn	Multipliziere mit Vorzeichen	muls #5,d0 muls (a0),d0
MULU ea,Dn	Multipliziere ohne Vorzeichen	mulu #5,d0 mulu (a0),d0
NEG ea	Negiere (ea=0-ea)	neg d0 neg (a0)
- Nicht für Adreßregister		

ADDQ und SUBQ dürfen, im Gegensatz zu MOVEQ, auch auf Adreßregister oder Speicheradressen angewandt werden. Die Konstante darf aber nur noch zwischen 0 und 8 liegen.

Die Divisions-Befehle schreiben das Ergebnis in das untere Wort des Langwort-Datenregisters (Bits 0-15) und den Rest in das obere (Bits 16-31). Zum Transport des Rests ins untere Wort ist der SWAP-Befehl gut geeignet.

Computer-interne Darstellung negativer Zahlen

An dieser Stelle ein kleiner Einschub: Speziell im Zusammenhang mit den Rechenbefehlen ist es interessant zu wissen, wie negative Zahlen computer-intern dargestellt werden.

Der Computer verwendet kein gesondertes Zeichen zur Darstellung negativer Zahlen, wie wir das beim Rechnen für gewöhnlich tun. Statt dessen erklärt er einfach das höchste Bit einer Zahl, die negative Werte annehmen können soll, zum "Vorzeichenbit". Wenn dieses Bit gesetzt ist, gilt die Zahl als negativ, ansonsten als positiv. Dabei ist natürlich der

Datentyp (Byte, Wort oder Langwort) der Zahl wichtig, da ja festgelegt werden muß, welches Bit nun das höchste ist.

Man könnte nun annehmen, daß die (Byte-)Zahl -3 vom Computer binär als %10000011 dargestellt wird, also einfach als binäre 3 mit zusätzlich gesetztem höchstem Bit. Dem ist aber nicht so! Der Computer verwendet stattdessen ein Verfahren, daß sich "Zweier-Komplement-Darstellung" nennt. Hinter diesem kompliziert klingenden Namen steckt folgendes System:

Soll eine Zahl negativ sein, so nimmt der Computer zunächst einmal die positive Form dieser Zahl in der Binärdarstellung. Für unsere -3 wäre das also %00000011. Nun wird diese Zahl komplementiert, d.h. alle Bits werden in ihr Gegenteil verkehrt. Aus %00000011 wird demnach %11111100. Damit haben wir schon die Bedingung, daß das höchste Bit gesetzt sein muß, erfüllt. Nun wird zu der Zahl noch eine binär-1 hinzuaddiert. Das bringt, wie wir gleich sehen werden, Vorteile beim internen Rechnen. Aus %11111100 wird dann also %11111101. Das ist die interne Darstellung der Zahl -3 (als Byte).

Die Verwendung negativer Zahlen bringt eine Einschränkung im Zahlenbereich mit sich: Ein Byte kann normalerweise Werte von 0 bis 255 annehmen. Bei einem Byte, das auch negativ sein kann, geht das höchste Bit für die Zahl selbst verloren, es sind also nur noch Werte von 0 bis 127 möglich. Bei den weiteren 128 Werten ist das höchste Bit gesetzt, sie repräsentieren also die Zahlen -128 bis -1. Der Wertebereich des Datentyps wird bei Verwendung negativer Zahlen also in positive Zahlen (einschließlich 0) und negative Zahlen aufgeteilt. Ein Wort (gewöhnlich 0-65535) kann Werte von 0-32767 und -32768 bis -1 annehmen.

Nun ein Rechenbeispiel in der computer-internen Darstellung. Die Subtraktion kann man als Addition mit umgekehrtem Vorzeichen ansehen: 50 - 29 entspricht 50 + (-29). Die Darstellung im Zweierkomplement führt zu folgender Binär-Rechnung:

```
Dez-Zahl     Binär-Zahl    Zweier-Komplement
--------------------------------------------
     50        00110010             00110010
  -  29      - 00011101           + 11100011
   -----     ----------          -----------
     21        00010101            100010101
                                   |
    Übertrag wird nicht beachtet --+
```

Der Übertrag im Zweierkomplement-Ergebnis wird nicht beachtet. Die restliche Binärzahl ergibt in dezimal genau die 21. Das Rechnen mit negativen Zahlen wird auf diese Weise recht einfach. Nun noch ein Beispiel, bei dem das Ergebnis negativ ist:

Dez-Zahl	Binär-Zahl	Zweier-Komplement
23	00010111	00010111
- 37	- 00100101	+ 11011011
- 14	11110010	11110010

Vorzeichen-Bit gesetzt

Das Vorzeichen-Bit im Ergebnis ist gesetzt, es handelt sich hierbei also um eine negative Zahl im Zweier-Komplement. Zur Rückumwandlung zieht man zunächst eine 1 ab (aus %11110010 wird %11110001) und komplementiert dann (aus %11110001 wird %00001110). Das Ergebnis ist die 14.

Die Zweierkomplement-Darstellung vereinfacht also das Rechnen mit Zahlen für den Computer stark. Er kann positive und negative Zahlen nach dem selben Verfahren addieren und subtrahieren, wobei ein eventueller Übertrag, der den Wertebereich überschreitet, einfach ignoriert wird.

2.5.4 Programmsteuer-Befehle

Zu dieser Gruppe zählen alle Verzweige-Befehle sowie die Vergleichs-Befehle und einige "Sonderlinge", die sich sonst nirgendwo einordnen lassen.

Befehl	Bedeutung	Beispiele
BRA Label - PC-relativer Sprung	Verzweige zu Label	bra markel
BSR Label - PC-relativer Sprung	Verzweige zu Unterprg.	bsr uprg1
Bcc Label - Siehe Anmerkungen	Verzw., wenn cc erfüllt	beq markel bne marke2
CMP ea,Dn - Siehe Anmerkungen	Vergleiche mit Datenregister	cmp #1,d0 cmp (a0),d0
CMPA ea,An	Vergleiche mit Adreßregister	cmp #1,a0 cmp (a0),a1
CMPI #k,ea	Vergleiche Konstante mit ea	cmp #1,1000 cmp #1,d0 cmp #1,a1
TST ea - Entspricht CMP #0,ea	Vergleiche mit 0	tst d0 tst (a0)

BTST Dn,ea BTST #k,ea - Siehe Anmerkungen	Teste Bit Dn o. #k von ea	btst d0,1000 btst #1,1000
DBcc Dn,Label - Siehe Anmerkungen	Schleife mit Abbruchsbed.	dbra d1,marke1 dbeq d2,marke2
JMP ea	Verzweige Absolut	jmp $fc00d2 jmp (a0)
JSR ea	Verzweige zu Unterprg.	jsr -198(a6) jsr (a0)
RTS	Rücksprung aus Unterprg.	rts
NOP - Nützlich als Füllbefehl beim Debuggen	No Operation - tue nichts	nop

Die Branch-Sprungbefehle (BRA, BSR, BCC usw.) sind PC-relativ, d.h. sie springen nicht zu einer absoluten Adresse, sondern erhöhen oder erniedrigen den PC um den entsprechenden Wert. Das hat zur Folge, daß die Sprungdistanz höchstens 32767 Bytes vorwärts oder 32768 Bytes rückwärts betragen darf. Durch die (alleinige) Verwendung solcher Sprungbefehle bleibt Ihr Programm positionsunabhängig. Die Jump-Sprungbefehle (JMP, JSR) dagegen arbeiten mit absoluten Adressen.

Nun kommen wir zur Realisierung von Abfragen und bedingten Sprüngen in Assembler. Dafür sind die CMP-, TST- und BTST-Befehle (Abfrage) und die Befehle BCC und DBCC zuständig.

2.5.5 Vergleichs-Befehle

Im Abschnitt über das Registermodell haben wir schon etwas über das Statusregister und die Flags erfahren. Dieses Wissen kommt nun zur Anwendung.

Der Haupt-Vergleichsbefehl ist der CMP-Befehl. Dieser führt im Grunde eine normale Subtraktion durch, schreibt das Ergebnis allerdings nirgendwo hin, sondern setzt nur die entsprechenden Flags. Die Abfrage "vergleiche d0 und d1" hieße dann also:

```
cmp.l   d0,d1
```

Dabei wird d1 minus d0 gerechnet. Wenn d0 gleich d1 ist, kommt bei der Subtraktion 0 heraus, das Z-Flag wird also gesetzt und kann im weiteren Verlauf ausgewertet werden. War d1 kleiner als d0, wird das Ergebnis negativ und das N-Flag gesetzt.

Genau wie bei den SUB-Befehlen wird hier der Quell-Operand vom Ziel-Operanden abgezogen. "Vergleiche d0 mit 5" müßte also heißen:

```
cmp.l   #5,d0
```

Eine Kurzform des Befehls "Vergleiche mit 0" ist der TST-Befehl. Um zu testen, ob d0 auf 0 steht, ist also folgendes möglich:

```
tst.l   d0
```

Das entspricht

```
cmp.l   #0,d0
```

Um ein einzelnes Bit zu testen, verwendet man den BTST-Befehl. Um zu prüfen, ob das 5. Bit von d0 auf 0 steht, schreibe ich:

```
btst    #5,d0
```

Bedingte Sprünge

Jetzt wissen wir, daß die Flags immer entsprechend dem Ergebnis des letzten Vergleichs-Befehls gesetzt sind. Um nun in Abhängigkeit von den Flagzuständen zu verzweigen, benutzen wir den BCC-Befehl. Das CC steht für "Condition Code", also Bedingungscode.

Kürzel	Bedeutung	Flag-Abfrage
CC	Carry Clear (Kein Übertrag)	c
CS	Carry Set (Übertrag)	C
EQ	Equal (Gleich)	Z
GE	Greater or Equal (>=)	nv + NV
GT	Greater Than (>)	nvz + NVz
HI	Higher (>)	cz
LE	Less or Equal (<=)	Z + Nv + nV
LS	Less or Same (<=)	C + Z
LT	Less Than (<)	Nv + nV
MI	Minus (Kleiner 0)	N
NE	Not Equal (Ungleich)	z
PL	Plus (Größer 0)	n
VC	Overflow Clear (Kein Überlauf)	v
VS	Overflow Set (Überlauf)	V

Bild 2.4: Die Condition Codes

Für uns als angehende Programmierer sind eigentlich nur die Kürzel und ihre Bedeutungen wichtig. Die Flag-Abfragen, also die Angaben, welche Flags beim Test worauf geprüft werden,

sind nur der Vollständigkeit halber (und zum Nachschlagen für fortgeschrittenen Programmierer) aufgeführt. Ein großer Buchstabe bedeutet hier, daß das entsprechende Flag auf 'gesetzt' getestet wird, bei einem kleinen wird auf 'nicht gesetzt' geprüft. Stehen mehrere Buchstaben direkt hintereinander, werden alle diese Flags getestet. Das '+' bedeutet 'oder' (Boolesche Algebra) und steht für Alternativen beim Test. Die Flag-Abfrage für 'Greater Than' (nvz + NVz) bedeutet also, daß entweder das N-, Z- und V-Flag gelöscht oder das N- und V-Flag gesetzt und das Z-Flag gelöscht sein muß.

Der Code "EQ" (steht für "Equal") prüft z.B, ob die Operanden des letzten Vergleichs gleichwertig waren. Effektiv wird geprüft, ob die CMP-Subtraktion 0 ergab, das Z-Flag also gesetzt ist.

Nun können wir für unseren bedingten Sprung den Bedingungscode auswählen, den wir brauchen. Wollen wir z.B. zur Marke 'marke1' springen, wenn der letzte Vergleich ergeben hat, daß die beiden Operanden ungleich waren, schreiben wir:

```
        bne     marke1
```

"BNE" bedeutet "Branch if Not Equal" - Verzweige, wenn nicht gleich. Ein weiteres Beispiel:

```
        bgt     marke2
```

Es wird verzweigt, wenn der Zieloperand des letzten Vergleichs größer war als der Quelloperand.

IF-Abfragen in Assembler

Da sich die BCC-Befehle auf den Zustand der Flags beziehen und diese durch fast alle Assembler-Befehle verändert werden, ist es wichtig, die zusammengehörigen Vergleichs- und Sprungbefehle auch direkt hintereinander zu schreiben. Ansonsten, wenn noch andere Befehle dazwischen kommen, könnten sich die Flags ja schon wieder verändert haben und beim Sprungbefehl dann nicht mehr das Ergebnis des Vergleichs, sondern irgendeines anderen Befehls darstellen. Daher besteht eine Assembler-IF-Abfrage immer aus zwei Befehlen: dem CMP (oder Vergleichbarem) und dem BCC. Einige Beispiele:

```
        cmp.l   #5,d0
        beq     marke1
```

Wenn d0 gleich 5 ist, wird zu 'marke1' verzweigt.

```
        cmp.l   d2,d3
        bgt     marke2
```

Wenn d3 größer als d2 ist, wird zu 'marke2' verzweigt.

```
        tst.l   d1
        bne     marke3
```

Es wird zu 'marke3' verzweigt, wenn d1 ungleich 0 ist.

Sehr wichtig ist hier, ebenso wie bei den Rechen-Befehlen, daß ein Vergleich immer in der Form "Ziel minus Quelle" durchgeführt wird. Außerdem müssen Sie im Hinterkopf behalten, daß die Verzweige-Befehle immer aufgrund der Flags reagieren. Welche Flags dabei wie genau getestet werden, ist für uns unwichtig, wir müssen nur den richtigen Condition Code auswählen.

Die Sonderform der bedingten Verzweigung, der DBCC-Befehl, dient der Programmierung von Schleifen. Mit ihm werden wir uns im 3. Kapitel eingehend beschäftigen.

2.5.6 Logische Befehle

Um die Bedeutung dieser Befehlsgruppe verstehen zu können, müssen wir erst einen kleinen Abstecher in die Boolesche Algebra machen. Das klingt jetzt vielleicht ein bißchen sehr nach Schulmathematik, aber so schlimm ist es gar nicht. Die Boolesche Algebra, eingeführt von einem gewissen Herrn Boole, befaßt sich mit bestimmten Regeln zur Verknüpfung von Binärzahlen.

Bei den booleschen Funktionen handelt es sich um Rechenoperationen, ähnlich wie Addition, Subtraktion usw. Ebenso, wie man z.B. 13 + 7 = 20 schreiben kann, könnte eine Gleichung mit boolescher Funktion lauten:

13 AND 9 = 5

Wie kommt dieses seltsame Ergebnis nun zustande? Wie gesagt, beziehen sich boolesche Funktionen, wie die AND-Funktion, auf Binärzahlen. Man müßte also 13 und 7 erstmal in die binäre Schreibweise übertragen:

13 = %1101   7 = %0111

Erinnern Sie sich? Das '%'-Zeichen war die Kennzeichnung für Binärzahlen, so wie das '$'-Zeichen für Hex-Zahlen.

Mit der AND-Funktion, auch "AND-Verknüpfung" genannt, hat es folgendes auf sich: Die beiden Zahlen werden Bit für Bit verglichen. Das entsprechende Bit in der Ergebniszahl ist nur dann 1, wenn die Bits beider Eingangszahlen an dieser Stelle ebenfalls 1 waren. Beispiel:

```
    13    % 1101
AND  7    % 0111
----------------
Ergebnis  % 0101   = 5
```

Daher kommt also die 5. Sie dürfen AND nicht mit plus verwechseln. Neben der AND-Verknüpfung gibt es noch einige andere. Die OR-Verknüpfung z.B. liefert 1, wenn eins der beiden Eingangsbits oder beide 1 waren.

Die booleschen Funktionen stellt man gewöhnlich anhand von "Wahrheitstabellen" dar. Dieser Ausdruck kommt daher, daß ein 0-Bit auch als "logisch falsch" und ein 1-Bit als "logisch wahr" bezeichnet wird. In einer solchen Tabelle stellen die Zeilen und Spalten die Eingangsbits dar, und das Ergebnis ist im "Innenraum" der Tabelle abzulesen. Hier nun die Wahrheitstabellen der vier wichtigsten logischen Verknüpfungen:

AND	**0**	**1**
0	0	0
1	0	1

OR	**0**	**1**
0	0	1
1	1	1

NOT	**0**	**1**
	1	0

EOR	**0**	**1**
0	0	1
1	1	0

Bild 2.5: Die Wahrheitstabellen für AND, OR, NOT und EOR

Die NOT-Verknüpfung hat nur einen Eingangswert und dreht alle Bits dieses Wertes um. Beispiel:

NOT 11000110 (=198) = 00111001 (=57)

Die EOR-Verknüpfung (Exklusiv-OR) unterscheidet sich dadurch von OR, daß sie eine 0 liefert, wenn beide Eingangsbits gesetzt sind. OR liefert hier eine 1.

Es gibt noch zwei weitere logische Verknüpfungen, die Implikation (IMP) und die Äquivalenz (EQV). Diese werden jedoch nicht durch MC68000-Befehle abgedeckt und werden auch so gut wie nie gebraucht.

So vorbereitet können wir uns nun an die Tabelle mit den logischen Befehlen des MC68000 wagen. Wie schon für die Rechenbefehle gilt hier:

```
        and.l    d0,d1
```

bedeutet: d1 AND d0, speichern in d1.

Befehl	Bedeutung	Beispiele
AND ea,ea	Logische UND-Verknüpfung	and #1,d0 and d0,(a0) and 1000,d0
ANDI #k,ea	Logisch UND mit Konstante	and #2,d0 and #4,(a0)
EOR ea,ea	Logische EOR-Verknüpfung	eor #1,d0 eor d0,(a0) eor 1000,d0
EORI #k,ea	Logisch EOR mit Konstante	eor #2,d0 eor #4,(a0)
NOT ea - Entspricht nicht neg!	Logische NOT-Verknüpfung	not d0 not 1000
OR ea,ea	Logische OR-Verknüpfung	or #1,d0 or d0,(a0) or 1000,d0
ORI #k,ea	Logisch OR mit Konstante	or #2,d0 or #4,(a0)

Der NOT-Befehl entspricht nicht, wie man vielleicht meinen könnte, dem NEG-Befehl. NEG bildet das Zweierkomplement der Zahl, während NOT lediglich alle Bits umkehrt.

2.5.7 Bit-Befehle

Diese Befehle dienen der Bearbeitung von Zahlen auf Binär-Ebene. Sie umfassen Schiebe-, Rotations- und Setz/Lösch-Befehle.

Die Schiebe- und Rotationsbefehle gibt es mit einem und zwei Parametern. Bei der Ein-Parameter-Form wird die Zahl im Zieloperanden immer um ein Bit geschoben, bei zwei Parametern gibt die Quelle an, um wieviel Bits die Zahl im Ziel geschoben werden soll. Die Zwei-Parameter-Form erlaubt aber nur Datenregister als Ziel.

Befehl	Bedeutung	Beispiele
ASL Dn,Dn ASL #k,Dn ASL ea - Siehe Anmerkungen	Arithm. Linksschieben	asl d0,d1 asl #2,d1 asl 1000

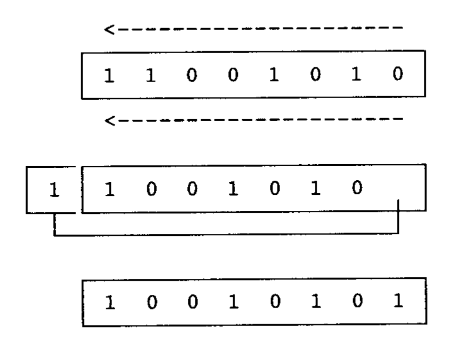
ASR Dn,Dn ASR #k,Dn ASR ea - Siehe Anmerkungen	Arithm. Rechtsschieben	asr d0,d1 asr #2,d1 asr 1000
BCHG Dn,ea BCHG #k,ea - Alter Zustand des Bits kommt ins Z-Flag	Ändere Bit Dn o. #k von ea	bchg d0,1000 bchg #5,1000
BCLR Dn,ea BCLR #k,ea - Alter Zustand des Bits kommt ins Z-Flag	Lösche Bit Dn o. #k von ea	bclr d0,1000 bclr #5,1000
BSET Dn,ea BSET #k,ea - Alter Zustand des Bits kommt ins Z-Flag	Setze Bit Dn o. #k von ea	bclr d0,1000 bset #5,1000
LSL Dn,Dn LSL #k,Dn LSL ea - Siehe Anmerkungen	Logisch Linksschieben	lsl d0,d1 lsl #2,d1 lsl 1000
LSR Dn,Dn LSR #k,Dn LSR ea - Siehe Anmerkungen	Logisch Rechtsschieben	lsr d0,d1 lsr #2,d1 lsr 1000
ROL Dn,Dn ROL #k,Dn ROL ea - Siehe Anmerkungen	Linksrotieren	rol d0,d1 rol #2,d1 rol 1000
ROR Dn,Dn ROR #k,Dn ROR ea - Siehe Anmerkungen	Rechtsrotieren	ror d0,d1 ror #2,d1 ror 1000

Nun zu den zwei Untergruppen der Bitbefehle:

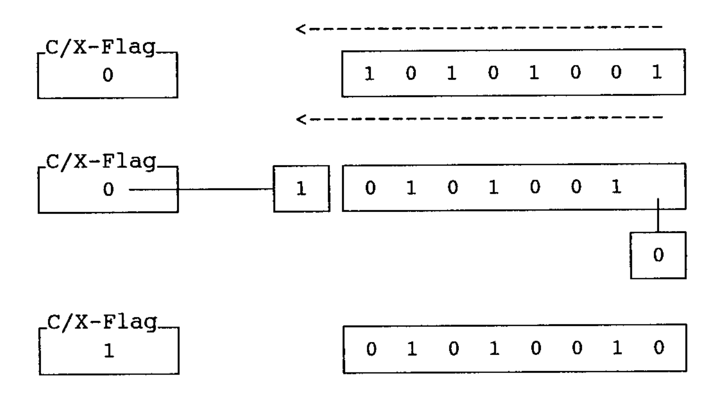
Schiebe- und Rotationsbefehle

Hier werden sämtliche Bits einer Binärzahl um eine festgelegte Anzahl Stellen nach links oder rechts geschoben.

Bei den Rotationsbefehlen wird das Bit, das links oder rechts herausfällt, an der anderen Seite wieder eingespeist. Beispiel: Die Binärzahl %11001010 soll nach links rotiert werden:



Die Schiebe-Befehlen verhalten sich etwas anders. Man unterscheidet hier zwischen arithmetischem und logischem Schieben. Ein Unterschied tritt aber nur beim Rechtsschieben auf, daher betrachten wir zunächst das Linksschieben. Hierbei wird das links herausgeschobene Bit nicht wieder eingespeist, es wird aber ins C- und X-Flag eingetragen. Rechts wird ein 0-Bit nachgeschoben. Beispiel: Die Binärzahl %10101001 wird arithmetisch oder logisch nach links geschoben:



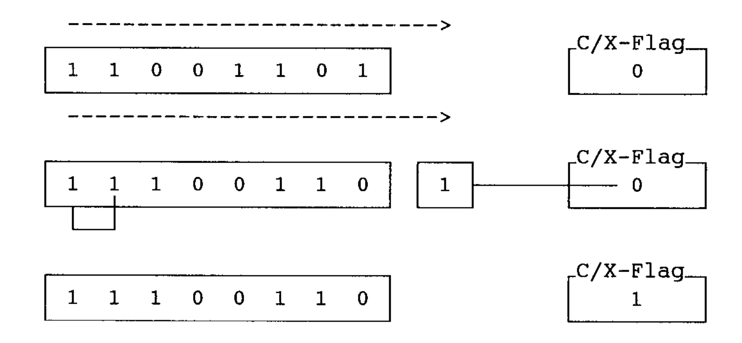
Mit dem ASL-Befehl läßt sich übrigens eine Multiplikation um 2, 4, 8 usw. (also mit 2er-Potenzen) sehr viel schneller erledigen als mit dem MULU-Befehl. Genauso, wie im Zehnersystem eine Verschiebung nach links einer Multiplikation mit 10, 100, 1000 usw. entspricht, bewirkt die Verschiebung einer Binärzahl nach links eine Multiplikation mit 2, 4, 8 usw. Anstatt

```
        mulu    #4,d0
```

schreibt man also besser

```
asl     #2,d0
```

Nun zum Unterschied zwischen arithmetischem und logischem Rechtsschieben. Das logische Rechtsschieben läuft analog zum Linksschieben (herausfallendes Bit ins C- und X-Flag, 0-Bit nachschieben). Beim arithmetischen Rechtsschieben wird auch das herausfallende Bit ins C- und X-Flag eingetragen. Es wird aber kein 0-Bit nachgeschoben, sondern das frühere Bit bleibt, obwohl es nach rechts geschoben wurde, auch an seiner alten Position erhalten. Es wird also quasi eine Stelle nach rechts "kopiert". Ein Beispiel: Die Binärzahl %11001001 wird arithmetisch nach rechts geschoben:



Bitmanipulations-Befehle

Diese Gruppe umfaßt die Befehle "Bit setzen" (BSET), "Bit löschen" (BCLR) und "Bit wechseln" (BCHG). Letzterer macht aus einem gesetzten Bit ein gelöschtes und umgekehrt. Ziel kann entweder ein Datenregister oder eine Adresse im RAM sein, die Quelle gibt die Nummer des zu bearbeitenden Bits an. Falls das Ziel im RAM liegt, ist als Nummer nur 0-7 zugelassen, d.h. die Adresse darf nur als Byte angesprochen werden. Bei Datenregistern hingegen sind alle Bitnummern des Langworts (0-31) erlaubt.

Soll z.B. das 5. Bit des Registers d0 gelöscht werden, schreibt man:

```
bclr    #5,d0
```

So setzt man das 'd0-te' Bit von d1:

```
bset    d0,d1
```

Und schließlich eine Bitmanipulation im RAM:

```
bclr    #5,1000         ; Löscht Bit 5 von Speicherstelle 1000
bchg    d0,1000         ; Wechselt das 'd0-te' Bit
```

Kapitel 3
Einstieg in Assembler

Aufbau eines Assembler-Programms

Assembler-Direktiven und Variablen

Einführung des Library-Konzepts

Erste Schritte in DOS

Die Kommandozeile

Schleifen-Techniken

Die ANSI-Steuerkommandos

Umrechnung von Zahlensystemen

Unterprogramme

Mehrfachverzweigungen

Abschluß-Programm: CLI-Taschenrechner

3.1 Aufbau eines Assembler-Programms

Ein Assemblerprogramm besteht, wie Programme in anderen Sprachen auch, aus mehr oder weniger vielen Befehlen. Im Gegensatz zu manchen anderen Sprachen dürfen Sie in Assembler aber immer nur einen Befehl pro Zeile schreiben. Eine Assembler-Befehlszeile besteht aus vier Teilen: dem "Label" (zu Deutsch "Marke"), dem Befehl, den Operanden und dem Kommentar. Label und Kommentar müssen nicht sein, und auch der Operand kann, je nach Befehl, wegfallen. Befehle und Operanden kennen wir ja schon zur Genüge, deshalb wollen wir uns jetzt den neuen Dingen zuwenden.

3.1.1 Kommentare

Da ein Assembler-Programm vor allem für Außenstehende meist recht schwer nachzuvollziehen ist, haben Sie die Möglichkeit, Ihre Quelltexte beliebig mit erklärenden Kommentaren zu versehen. Wie das geht, haben Sie bestimmt schon in diversen Beispiel-Befehlszeilen gesehen: Sie schreiben hinter Befehl und Operanden ein Semikolon, und alles, was danach noch bis zum Zeilenende folgt, gilt als Kommentar. Beispiel:

```
        move.l   d0,a0          ; <- Semikolon = Kommentarzeichen
```

Soll ein Kommentar alleine in einer Zeile stehen, beginnen Sie diese einfach mit einem Stern:

```
* <- Stern = Zeichen für Nur-Kommentar-Zeile
```

3.1.2 Labels

Wann immer Sie zu einer bestimmten Stelle in Ihrem Programm verzweigen wollen, sei es ein Unterprogramm, ein bedingter oder unbedingter Sprung, müssen Sie die Adresse des Befehls kennen, zu dem Sie springen möchten (denn Verzweigung heißt für die CPU: "Lade den Programmzähler mit der Adresse des anzuspringenden Befehls"). Das hieße aber, daß Sie sämtliche Befehle des Programms quasi "durchzählen" müßten, um die Adressen aller Befehle zu wissen. Das wäre eine unglaubliche, um nicht zu sagen unmögliche, Arbeit. Als Abhilfe dafür bietet der Assembler die Labels (Marken) an. Sie stellen einfach vor jeden Befehl, den Sie im Laufe des Programms anspringen oder sonstwie ansprechen möchten, ein Label. Dieses repräsentiert dann für den Assembler die Adresse des Befehls.

Ein Label besteht einfach aus einem Text, den Sie sich (mehr oder weniger) frei auswählen können. Die meisten Assembler machen hier die Auflage, daß Labels mit einem Buchstaben beginnen müssen und nur Buchstaben und Zahlen enthalten dürfen. Um einen Text als Label zu kennzeichnen, müssen Sie bei

manchen Assemblern einen Doppelpunkt an ihn anschließen, bei manchen reicht es, wenn er in der ersten Spalte der Befehlszeile beginnt. Ein paar Beispiele für Label-Anwendungen:

```
        tst.b   d0          ; Ist d0=0?
        beq     marke       ; Wenn ja, springe zu marke
        add.b   #5,d0       ; Ansonsten addiere 5 zu d0
marke:  move.b  d0,d4       ; Zur Weiterverarbeitung
```

```
        move.l  a4,a0       ; Eine Beispieladresse (z.B. ein
                            ; Tabellenstart)
marke:  move.b  (a0)+,d0    ; Erstes Tabellen-Byte holen
                            ; und Tabellenzeiger plus 1
        cmp.l   a3,a0       ; Tabellenende (soll in a3
                            ; stehen) erreicht?
        blt     marke       ; Solange Zeiger kleiner als
                            ; Ende, Rücksprung zu marke
```

Bild 3.1: Beispiele für Label-Anwendungen

3.1.3 Wichtige Assembler-Direktiven

Eine Assembler-Direktive ist auch ein Befehl, allerdings kein Befehl des MC68000. Vielmehr ist es eine Anweisung direkt an das Übersetzungsprogramm. Die wichtigsten Direktiven werden wir nun kennenlernen.

Die 'equ'-Direktive

Die equ-Direktive bedeutet schlicht "setze gleich" (equal). Auf diese Weise kann man irgendwelchen Zahlenwerten (Tabellenplatznummern, Adressen etc.) sinnvolle Namen zuordnen, die sich bestimmt leichter merken lassen. Der Befehl

```
TabPlatz      equ     20
```

setzt also dem Text 'TabPlatz' den Wert 20 gleich. Im Grunde ist das reine Textverarbeitung, der Assembler setzt später einfach überall da, wo TabPlatz steht, eine 20 ein. Prinzipiell sind diese equ-Zuweisungen nicht unbedingt notwendig, sie helfen Ihnen aber, wenn Sie sich Zahlen nur schlecht merken können (oder wollen). Sie können allen im Programm verwendeten Zahlenwerten und sonstigen Nummern sinnvolle Texte zuweisen.

Anstatt 'equ' kann man auch einfach ein Gleichheitszeichen schreiben:

```
TabPlatz      =       20
```

Die 'DC'- und die 'DS'-Direktive

Die 'DC'-Direktive dient zum Einfügen von beliebigen Zahlenwerten oder Texten in ein Programm. Wenn Sie z.B. den Text "Hello World!" in Ihrem Programm ausgeben wollen, muß er ja auch irgendwo stehen. Sie können dann schreiben:

```
        dc.b    "Hello World!"
```

Damit wird der Text an der Stelle, wo die Zeile im Programm auftritt, in das übersetzte Maschinenprogramm eingefügt. Natürlich kann man auch Zahlen mit 'DC' ins Programm einbauen, oder auch Text und Zahlen gemischt:

```
        dc.b    "Hello World!",10,"How are you?",10
```

Die 10 ist der ASCII-Code für 'neue Zeile'. Wenn man diesen Text auf dem Bildschirm ausgeben läßt, werden die beiden Sätze in getrennte Zeilen geschrieben. Sie können ASCII-Texte (in Anführungszeichen) oder Zahlen eines beliebigen Systems ('$' für Hex usw. voranstellen) beliebig mischen. Wichtig ist auch bei 'DC', den Datentyp der einzufügenden Werte anzugeben. So wird bei

```
        dc.b    0
```

ein 0-Byte ins Programm eingefügt, bei

```
        dc.l    0
```

aber ein 0-Langwort. Meistens muß man die Stelle von Texten oder Zahlen im Programm, also ihre Adresse, kennen. Daher findet man in der Regel eine Kombination aus Label und 'DC':

```
htext:  dc.b    "Hello World!",10
```

Nun zur 'DS'-Direktiven: Sie reserviert Speicherplatz im Programm, und zwar eine festlegbare Anzahl von Bytes, Worten oder Langworten. Beispiel:

```
puffer: ds.b    20
```

Dieser Befehl fügt ab der Label-Adresse 'puffer' 20 Bytes ins Programm ein. Achtung: Beim Seka-Assembler heißt diese Direktive 'BLK' und nicht 'DS'! Der Befehl

```
puffer2:        ds.l    20
```

fügt 20 Langworte (also 80 Bytes) ein. Dieser Befehl ist nützlich, wenn man im Programm Platz für Tabellen und ähnliche Strukturen braucht, die während des Programmlaufs erzeugt werden.

Sehr wichtig ist, daß man 'DC' und auch 'DS' nicht beliebig mitten ins Programm schreiben darf. Dann würde die CPU ja versuchen,, wenn sie diese Stelle erreicht, den Text o.ä.

als Befehl zu interpretieren, was sicherlich schief gehen dürfte. Am besten sammelt man alle 'DC'- und 'DS'-Direktiven am Programmende, hinter dem letzten Assembler-Befehl. Dies nennt man das Anlegen eines "Datenbereiches".

Außerdem müssen Sie beachten, daß es zwei Arten gibt, einen mit 'DC' oder 'DS' definierten Ausdruck "anzusprechen". Angenommen, Sie haben mittels

```
text:       dc.b    "Ausgabe-Text"
```

einen Text im Programm abgelegt. Dann können Sie durch

```
        move.l  #text,a0        oder
        lea     text,a0
```

die Startadresse des Textes nach a0 holen. Bei Verwendung von 'move.l' müssen Sie ein '#' verwenden, da sich der MOVE-Befehl ansonsten auf den Inhalt der Speicherstelle 'text' und nicht auf ihre Adresse beziehen würde. Beim LEA-Befehl ist kein '#' nötig, da er sowieso immer Adressen anspricht und keine Speicherinhalte. Wenn Sie jetzt aber durch

```
wert1:      dc.l    $44445555
```

ein Langwort im Programm stehen haben und Sie den Inhalt dieses Langworts bearbeiten wollen, müssen Sie z.B.

```
        move.l  wert1,d0
```

schreiben. Dieser MOVE-Befehl bezieht sich dann auf den Inhalt der Speicherstelle, während

```
        move.l  #wert1,d0       oder
        lea     wert1,a0
```

die Adresse holen würde (beachten Sie, daß LEA nur für Adreßregister zugelassen ist). Diese beiden "Ansprecharten" (Adresse oder Inhalt) müssen Sie gut auseinanderhalten.

Die 'even'-Direktive

Diese Direktive ist schnell erklärt: Sie sorgt dafür, daß das Programm mit dem nächsten Befehl (oder der nächsten Direktiven) auf jeden Fall an einer geraden Adresse fortgesetzt wird. Effektiv wird, falls eine ungerade Adresse erwischt würde, einfach ein 0-Byte (oder sonstwas) eingeschoben. Das ist nötig, weil Assembler-Befehle, aber auch bestimmte Daten, immer an geraden Adressen stehen müssen (sonst freut sich der Guru). In der Regel wird man diese Direktive im Datenbereich (oder halt da, wo Daten mit ungerader Länge stehen) finden.

Abhängig vom verwendeten Assembler gibt es ein paar Varianten der 'even'-Direktive. Beim Profimat heißt sie z.B.

'align' und nicht 'even'. Hier gibt es auch die Möglichkeit, auf eine Langwortgrenze (durch 4 teilbare Adresse) zu alignen: Man schreibt in diesem Fall 'align.l'. Das geht auch beim Devpac: Hier schreibt man 'cnop 0,4'.

3.1.4 Die 'include'-Direktive

'Include' weist den Assembler an, während der Übersetzung einen weiteren Quelltext, dessen Namen Sie hinter 'include' angeben, in Ihr Programm einzufügen. Dieser erscheint allerdings nicht in Ihrem Text, sondern wird nur während der Assemblierung eingeladen, mit in das fertige Programm hineingenommen und dann wieder vergessen. Diese Direktive ist nützlich, wenn Sie z.B. oft gebrauchte, immer gleichlautende Programmteile quasi "auslagern" möchten. Sie können den Programmteil in einer gesonderten Datei auf der Diskette halten und mit 'include' in Ihre Programme einlesen lassen, ohne daß er im Quelltext selber auftaucht. Das spart Platz im Textspeicher und macht die Programme übersichtlicher.

Außerdem liefern die meisten Assembler einen Satz sogenannter "Include-Files" mit. Darunter versteht man eine Reihe von Textdateien, die Sie in Ihre Programme einbinden können. Die Include-Files bestehen hauptsächlich aus 'equ'- und sonstigen Direktiven und sind nach Themengebieten geordnet. Die DOS-Include-Files enthalten Zuweisungen für Zahlen, die man häufig im Umgang mit Dateien braucht, die Intuition-Include-Files entsprechend 'equ's für Screen- und Windowprogrammierung.

Manche Programmierer machen intensiven Gebrauch von diesen Includes. Das bringt aber auch einen Nachteil mit sich: Obwohl man meistens nur einen kleinen Teil des Includes braucht, muß man trotzdem immer die ganze Datei einladen. Außerdem sind die Include-Files weitgehend untereinander verkettet, d.h. ein Include lädt selbst wieder diverse andere ein, diese wieder weitere usw. Obwohl Sie vielleicht nur ein oder zwei Werte aus einem Intuition-Include haben wollten, werden vielleicht plötzlich noch 3 oder 4 andere Intuition-Includes und von diesen noch 5 bis 6 Exec-Includes nachgeladen. Das bringt natürlich eine recht lange Assemblierzeit mit sich (vor allem, wenn Sie mit Disketten arbeiten), und außerdem wird eine Menge Speicherplatz benötigt.

Wir halten es persönlich für sinnvoller, die Includes quasi als "Nachschlagewerke" zu benutzen. Wir suchen uns die Zahlenwerte, die wir brauchen, aus den Includes zusammen und schreiben sie selber per 'equ' in unsere Programme. Bei den wenigen Zahlen, die man als Einsteiger braucht, ist das auf jeden Fall günstiger. Später, wenn man als Fortgeschrittener nach einem bestimmten Wert oder einer Systemstruktur sucht, können sich die Includes als wahre Fundgrube erweisen.

3.1.5 Variablen

Mit unserem jetzigen Wissen ist es kein Problem mehr, Variablen in unsere Programme einzubauen. Eine Variable ist im Grunde (in jeder Programmiersprache) ein Speicherbereich von bestimmter Länge, der über einen Namen angesprochen wird. Sprachen wie BASIC nehmen uns die Verwaltung dieses Speicherbereichs ab, in Assembler müssen wir allerdings selbst dafür sorgen. Ein Beispiel: Wollen wir eine Langwort-Variable namens 'zahl1' benutzen, schreiben wir folgendes in den Datenbereich des Programms:

```
zahl1:        ds.l    1
```

Das bedeutet: Reserviere ein Langwort an dieser Stelle im Programm und gebe seiner Adresse den symbolischen Namen 'zahl1'. Sie sehen es vielleicht schon: Der Variablenname ist in Assembler gleich dem Label, das wir vor den Reservierungs-Befehl schreiben. Ein anderes Beispiel:

```
zahl2:        ds.w    1
```

reserviert ein Wort Platz unter dem Namen 'zahl2'. Auch Tabellen, Arrays u.ä. lassen sich so erstellen:

```
tabelle1:     ds.b    30
```

erstellt eine Tabelle mit 30 Bytes, deren Startadresse das Label 'tabelle1' darstellt.

3.2 Libraries: Grundlage des Amiga-Systems

Bis jetzt haben wir uns nur mit der Assemblersprache des MC68000 ganz allgemein beschäftigt. Alles, was wir bisher gelernt haben, ließe sich auch auf einen anderen Computer, der diese CPU besitzt, übertragen. Jetzt wollen wir etwas systemspezifischer werden und uns die wichtigste Einrichtung des Amiga-Systems vornehmen: die Libraries.

3.2.1 Was ist eine Library?

Im Grunde ist eine Library nichts weiter als eine Sammlung von Routinen, auf die Sie als Programmierer zurückgreifen können. Aber auch das Betriebssystem des Amiga macht regen Gebrauch von diesen Programmsammlungen, man kann sogar sagen, das System basiert auf den Libraries.

Die vier wichtigsten Libraries des Amiga sind

Exec
DOS

Intuition
Graphics

Die Exec-Library ist dabei der "Boß", sie ist im wesentlichen für das Multitasking (also die Verteilung der Computer-Rechenzeit an verschiedene, scheinbar gleichzeitig ablaufende Programme) und für die Zuteilung von freien RAM-Bereichen an die Tasks zuständig. Als Ottonormal-Anwender spüren wir von ihrer Arbeit allerdings gewöhnlich nicht viel.

Anders ist es da schon bei der DOS-Library. Wann immer Sie eine Datei umbenennen, ein Verzeichnis anlegen oder löschen oder auch nur ein Programm starten, die DOS-Library hat bestimmt ihre Routinen im Spiel. Sie ist für sämtliche Datei-Operationen zuständig, und das gilt nicht nur für Disk-Dateien. Die DOS-Library kann auch den Drucker ansteuern oder einfache Fenster öffnen, die dann auch wie Dateien behandelt werden.

Die Libraries Intuition und Graphics kommen immer dann ins Spiel, wenn es um Fenster, Menüs usw. (Intuition) oder um Linienzeichnen, Text usw. (Graphics) geht. Zu diesen beiden Libraries kommen wir später, zunächst einmal wollen wir uns mit der DOS-Library beschäftigen, da sie am einfachsten zu handhaben ist. Vorher müssen wir allerdings wissen, wie die Library-Benutzung überhaupt funktioniert.

3.2.2 Der Umgang mit Libraries

Eine Library ist, wie gesagt, eine Sammlung von Assembler-Routinen, die vom eigenen Programm aus wie Unterprogramme angesprungen werden können. Um aber ein Unterprogramm anspringen zu können, muß man dessen Adresse kennen. Nun darf man aber nicht alle Library-Routinen über absolute Adressen aufrufen, denn das würde heißen, daß sich nach jeder Änderung an der Library seitens Commodore die Adressen ändern würden und alle älteren Programme nicht mehr lauffähig wären.

Der Amiga benutzt eine recht zukunftssichere Methode des Libraryaufrufs. Neben den eigentlichen Routinen gibt es nämlich für jede Library eine Tabelle, in der die Einsprungadressen aller Routinen aufgeführt sind. Wenn man nun dem System mitteilt, das man eine bestimmte Library benutzen möchte (d.h. man 'öffnet' die Library), bekommt man die Startadresse dieser Sprungtabelle zurückgemeldet. Auf diese Startadresse (auch "Basisadresse" genannt) müssen sich dann alle Libraryzugriffe des Programms beziehen. Der Vorteil liegt auf der Hand: Solange sich die Position eines Sprungeintrages relativ zum Tabellenanfang nicht ändert (und das tut sie nie), kann die Library stehen, wo sie will, und sogar die einzelnen Routinen können nach Commodores Wünschen verschoben werden.

Was muß man denn nun tun, um eine Library zu öffnen? Ganz einfach: Man übergibt der Exec-Routine OpenLibrary den Namen (und eventuell die Versionsnummer) der gewünschten Library, und um den Rest kümmert sich dann das System. Das heißt, die Library wird, falls sie nicht schon geöffnet ist und auch nicht im ROM steht, von der Diskette geladen, und zwar in einen Speicherbereich, den Exec selbst auswählt. Das Programm erhält dann die Basisadresse mitgeteilt, und die Library steht zur Benutzung bereit. Falls nun noch ein weiteres Programm die selbe Library benutzen will, teilt ihm das System nur noch die Basisadresse mit.

Wichtig ist, daß man alle Libraries, die man geöffnet hat, vor dem Programmende auch wieder schließt, denn das System entfernt eine Library erst dann aus dem Speicher, wenn alle Programme gesagt haben, das sie sie nicht mehr brauchen. Dafür ist die Exec-Routine CloseLibrary zuständig.

Doch halt! werden Sie jetzt vielleicht denken. Um Routinen einer Library benutzen zu können, braucht man deren Basisadresse. Diese erhält man bei Aufruf der OpenLibrary-Routine, die ihrerseits in einer Library (der Exec-Library) steht. Woher weiß man denn die Basisadresse dieser Library? Des Rätsels Lösung lautet 4. In der Speicherstelle Nummer 4, die einzige wirklich feste Adresse im Amiga-System, findet man die Basis der Exec-Library. Diese braucht also nicht geöffnet (und auch nicht geschlossen) zu werden und ist jederzeit einsatzbereit.

Jetzt sind also alle nötigen Vorarbeiten geleistet, und die Library-Routinen können endlich benutzt werden. Die Basisadresse der Library trägt man in ein Adreßregister (gewöhnlich a6) ein, z.B.

```
move.l  4,a6
```

um die Exec-Library zu benutzen. Die benötigten Parameter kommen in die anderen Register; in welche, ist für jede Library-Routine festgelegt (z.B. erwartet OpenLibrary in a1 einen Zeiger auf den Namen der Library und in d0 die gewünschte Versionsnummer). Aufgerufen wird die Routine mittels

```
jsr     Offset(a6)
```

also über die Adressierungsart 'Adreßregister indirekt mit Offset'. Der Offset hier die Nummer der Routine in der Tabelle. Diese Offsets sind immer negativ, fangen bei -30 an und werden in Sechserschritten gezählt (warum das alles so ist, wird später noch geklärt). Die erste Routine der DOS-Library (genannt Open) hat also die Nummer -30, die zweite Routine (Close) die Nummer -36, die dritte (Read) -42 usw.

Jede Library-Routine (oder fast jede) erwartet, wie gesagt, bestimmte Parameter in Daten- und/oder Adreßregistern (ähnlich wie die Operanden eines Assembler-Befehls). Wenn

sie ein Ergebnis produzierte (wie z.B. die Library-Basisadresse bei OpenLibrary), steht diese nach dem Aufruf in d0.

Immer, wenn im Lauf des Kurses eine neue Library-Routine verwendet wird, stellen wir sie auf folgende Weise vor (als Beispiel die OpenLibrary-Routine):

OpenLibrary	=	**-552 (Exec-Library)**

***libName**	**a1**	<	Zeiger auf Library-Namenstext
version	**d0**	<	Versionsnummer (0=Version egal)
***library**	**d0**	>	Basis-Adresse oder 0 bei Fehler
Erklärung			Öffnet die angegebene Library

Zur Erklärung dieses Kastens: In der ersten Zeile steht der Name der Library-Routine und hinter dem '=' ihr Offset. In Klammern steht der Name der Library, aus der die Routine stammt. Unter dem Kasten stehen die Parameter mit den Registern, in die sie geschrieben werden müssen. Vor dem Register steht jeweils die englische Standard-Bezeichnung (gemäß Commodores Angaben) und dahinter die deutsche, etwas ausführlichere Erklärung. Das '<'-Zeichen steht für einen Parameter, der vor dem Aufruf eingesetzt werden muß. Das '>' kennzeichnet den Rückgabewert der Routine. Das '*' vor z.B. 'libName' bedeutet, daß der Name nicht direkt in a1 eingetragen wird (was wohl gar nicht möglich wäre), sondern daß in a1 ein "Zeiger" ("Pointer") auf den Library-Namen erwartet wird. Einen Stern verwenden wir in Anlehnung an die Sprache C, wo er auch das Zeichen für einen Pointer ist. In d0 hingegen wird direkt die Versionsnummer der Library eingetragen (dort steht kein '*'). Nach dem Aufruf bekommen wir in d0 die Basisadresse der Library bei erfolgreichem Öffnen oder 0 bei Fehler zurück. Falls eine Library-Routine keine Rückgabe liefert, fehlt die 'd0 >'-Zeile.

'Zeiger' bedeutet im Prinzip soviel wie 'Startadresse'. In unserem Beispiel muß also die Startadresse des (per 'DC' im Programm abgelegten) Library-Namenstextes in a1 geschrieben werden. Generell ist ein Zeiger immer die Startadresse irgendeines Objektes im Speicher. Bei unseren Library-Kästen setzen wir immer einen Stern vor einen Eintrag, wenn es sich dabei um einen Zeiger handelt. Dabei ist es egal, ob es sich um einen Zeiger auf ein Objekt (Struktur, Namenstext etc.) handelt, das der Programmierer angelegt hat oder das System bereitstellt.

Folgendermaßen würde ein Library-Aufruf im Programm aussehen:

```
        move.l   4,a6           ; Basis Exec
        move.l   #dosname,a1    ; Zeiger auf den Namen DOS-Library
        clr.l    d0             ; 0 = Version egal
        jsr      -552(a6)       ; Aufruf der OpenLibrary-Routine
        move.l   d0,dosbase     ; Basis DOS (Rückgabewert) sichern
        ...      ...

* Datenbereich

dosname:     dc.b    "dos.library",0
             even
dosbase:     ds.l    1
```

Bild 3.2: Beispielprogramm für Library-Routinenaufruf

Das Label 'dosname' wird per '#' angesprochen, da seine Adresse und nicht sein Inhalt gefordert ist. 'dosbase' ist eine Langwort-Variable, in die die von OpenLibrary gemeldete DOS-Basis zwischengespeichert wird.

3.3 Erste Schritte mit der DOS-Library

Nachdem wir nun das nötige "Handwerkszeug" beisammen und uns mit dem Library-Konzept vertraut gemacht haben, können wir mit der Programmierung loslegen. Wir hoffen, die bisherigen Theorie-Kapitel waren Ihnen nicht zu trocken, aber ein paar Grundlagen müssen halt sein. Alles weitere werden Sie ab jetzt, nach dem Motto "Learning by Doing", anhand von Programmen lernen.

3.3.1 Das erste Programm: Textausgabe im CLI-Fenster

Die nun folgenden Programme sollen als Grundlage für den weiteren Kurs dienen. Textausgabe, Eingabe über Tastatur usw. sind wichtig, damit wir die Ergebnisse unserer Programmierversuche auch am Bildschirm sehen können.

Bevor wir uns das Programm ansehen, brauchen wir noch ein paar weitere Library-Routinen, und auch ein paar Erklärungen sind fällig. Zuerst soll die CloseLibrary-Routine vorgestellt werden:

CloseLibrary	**=**	**-414 (Exec-Library)**

***library**	**a1**	<	Basisadresse der zu schließenden Library
Erklärung			Schließt die angegebene Library

Des weiteren brauchen wir ein "Handle" für die Standard-Bildschirmausgabe (das CLI-Fenster) geben. Das 'Handle' ist ein Begriff aus der DOS-Dateiverwaltung. Immer, wenn Sie eine Datei über DOS öffnen, bekommen Sie, quasi als "Zugriffsnummer" der Datei, eine Zahl, eben das Handle, mitgeteilt (vergleichbar mit der Filenummer bei BASIC). Bei allen DOS-Zugriffen wird die Datei dann über das Handle identifiziert.

CLI-Fenster werden im DOS auch als Dateien behandelt und haben daher auch ein Handle. Von der DOS-Routine Output lassen wir uns das Handle geben, das zum CLI-Fenster gehört, von dem aus wir unser Programm gestartet haben.

Output	=	**-60 (DOS-Library)**

***file**	**d0**	>	Handle des CLI-Fensters
Erklärung			Ermittelt das Standard-Output-Handle

Außerdem brauchen wir eine Routine, die einen Text in das Fenster schreibt. Da DOS-Fenster wie Dateien behandelt werden, können wir zu diesem Zweck die Standard-Daten-Schreibe-Routine verwenden.

Write	=	**-48 (DOS-Library)**

***file**	**d1**	<	Handle der zu schreibenden Datei
***buffer**	**d2**	<	Startadresse der Daten
length	**d3**	<	Anzahl Datenbytes
writtenOut	**d0**	>	Anzahl wirklich geschriebener Bytes (-1=Fehler)
Erklärung			Schreibt Daten in eine Datei

Nun haben wir alles beisammen, um das Programm schreiben zu können. Hier ist es:

```
* Programm 3.1: Textausgabe im CLI-Fenster

ExecBase    =    4
OpenLib     =    -552
CloseLib    =    -414
Write       =    -48
Output      =    -60
```

```
* DOS-Lib öffnen
        move.l   ExecBase,a6      ; Exec-Lib-Basis
        lea      dosname,a1       ; Name der DOS-Lib
        clr.l    d0               ; Version egal
        jsr      OpenLib(a6)      ; Lib öffnen
        tst.l    d0               ; Fehler beim öffnen?
        beq      ende             ; Wenn ja, zum Ende
        move.l   d0,dosbase       ; Basis DOS-Lib sichern

* Ausgabe-Handle des CLI-Fensters ermitteln:

        move.l   dosbase,a6       ; DOS-Basis nach a6
        jsr      Output(a6)       ; Output-Handle holen
        move.l   d0,clihandle     ; Handle sichern

* Text ausgeben:

        move.l   clihandle,d1     ; Handle nach d1
        move.l   #text,d2         ; Textbeginn nach d2
        move.l   #33,d3           ; Textlänge nach d3
        jsr      Write(a6)        ; DOS-Routine Schreiben

* Programmschluß: Lib schließen!

        move.l   dosbase,a1       ; Basis DOS-Lib
        move.l   4,a6             ; Exec-Basis
        jsr      CloseLib(a6)     ; Lib schließen

ende:   rts                       ; Zurück zum CLI

* Datenbereich: diverse Namen etc.

dosname:       dc.b    "dos.library",0
               even
dosbase:       ds.l    1
clihandle:     ds.l    1
text:          dc.b    "Mein erster Text im CLI-Fenster!",10
               even
```

Programm 3.1: Textausgabe im CLI-Fenster

Sie werden vieles, was Sie in den vorigen Kapiteln gelernt haben, hier wiederfinden. Da wäre zum einen das Sternchen am Anfang einiger Zeilen (Kommentar-Zeile) und das ';' für einen Kommentar im Anschluß an den Befehl.

Dann die Zuweisungen mit dem '='-Zeichen (Abkürzung für die 'equ'-Direktive). Hier sehen Sie den Hauptzweck von 'equ': Library-Offsets und sonstigen Zahlenwerten werden die entsprechenden Klartext-Bezeichnungen zugewiesen.

Im Datenbereich finden Sie die 'even'-Direktive wieder. Nochmal: Nach jedem 'DC'- oder 'DS'-Eintrag, von dem Sie

nicht ganz genau wissen, daß er eine gerade Bytezahl belegt, müssen Sie ein 'even' schreiben.

Dann wird per Exec-OpenLibrary die DOS-Library geöffnet. Wenn das nicht geklappt hat, steht d0 auf 0. Dies prüfen wir in den Zeilen

```
tst.l   d0          ; Fehler beim öffnen?
beq     ende        ; Wenn ja, zum Ende
```

Da haben wir also eine Assembler-IF-Abfrage, wie wir sie im letzten Kapitel kennengelernt haben. Im nächsten Abschnitt werden uns noch weitere IF-ELSE-Konstruktionen und Ähnliches begegnen.

Wenn die DOS-Lib geöffnet werden konnte, geht es weiter im Programm. Wir sichern die Basis der DOS-Lib in einer Variablen. Nun rufen wir, nachdem wir die DOS-Basis in a6 eingetragen haben, die Output-Routine auf. Von Output bekommen wir das Handle für die Standard-Bildschirmausgabe (das CLI-Fenster). In dieses Fenster geben wir den Text aus, und zwar so:

```
move.l  clihandle,d1    ; Handle nach d1
move.l  #text,d2        ; Textbeginn nach d2
move.l  #33,d3          ; Textlänge nach d3
jsr     Write(a6)       ; DOS-Routine Schreiben
```

Achten Sie auf das '#' in '#text'. Wie früher schon erwähnt, bewirkt es, daß die Adresse von 'text' und nicht sein Inhalt angesprochen wird.

Zum Abschluß des Programms müssen wir die DOS-Lib wieder schließen. Beendet wird das Programm mit

```
ende:   rts
```

damit zur DOS-Ebene zurückgekehrt wird.

Jetzt können wir uns an unserem ersten CLI-Text, von einem Assembler-Programm ausgegeben, erfreuen. Nachdem wir nun Text schreiben können, wäre es doch schön, auch etwas von der Tastatur einlesen zu können. Damit kommen wir zum zweiten Programm.

3.3.2 Texteingabe von Tastatur

Zu diesem Zweck brauchen wir zuerst zwei neue Library-Routinen. Zum einen die Input-Routine:

Input	=	**-54 (DOS-Library)**

***file** **d0** > Handle der Tastatur

Erklärung Ermittelt das Standard-Input-Handle

Die Tastatur (also das Standard-Eingabegerät) hat, genauso wie das CLI-Fenster, ein Handle. Neben der Write-Routine gibt es natürlich auch Read:

Read	=	**-42 (DOS-Library)**

***file** **d1** < Handle der zu lesenden Datei
***buffer** **d2** < Startadresse des Lese-Puffers
length **d3** < Anzahl Datenbytes

reallyread **d0** > Anzahl wirklich gelesener Bytes (-1=Fehler)

Erklärung Liest Daten aus einer Datei

So vorbereitet können wir uns an das Programm wagen:

```
* Programm 3.2: Eingabe über Tastatur

ExecBase      =      4
OpenLib       =      -552
CloseLib      =      -414
Write         =      -48
Read          =      -42
Output        =      -60
Input         =      -54

* DOS-Lib öffnen

        move.l   ExecBase,a6      ; Exec-Lib-Basis
        lea      dosname,a1       ; Name der DOS-Lib
        clr.l    d0               ; Version egal
        jsr      OpenLib(a6)      ; Lib öffnen
        tst.l    d0               ; Fehler beim öffnen?
        beq      ende             ; Wenn ja, zum Ende
        move.l   d0,a6            ; Basis DOS-Lib nach a6

* Output-Handle holen

        jsr      Output(a6)       ; Output-Handle holen
        move.l   d0,d5            ; Handle in d5 sichern
```

```
* Eingabeaufforderung ausgeben

        move.l   d5,d1             ; Output-Handle nach d1
        move.l   #text1,d2         ; Textbeginn nach d2
        move.l   #26,d3            ; Textlänge nach d3
        jsr      Write(a6)         ; DOS-Routine Schreiben

* Text über Tastatur einlesen

        jsr      Input(a6)         ; Input-Handle holen
        move.l   d0,d1             ; nach d1
        move.l   #buffer,d2        ; Start des Eingabe-Puffers
        move.l   #40,d3            ; 40 Zeichen max.
        jsr      Read(a6)          ; Lesen
        move.l   d0,d4             ; Anzahl gelesener Zeichen
                                   ; in d4 zwischenspeichern

* Text ausgeben

        move.l   d5,d1             ; Output-Handle nach d1
        move.l   #text2,d2         ; Textbeginn nach d2
        move.l   #23,d3            ; Textlänge nach d3
        jsr      Write(a6)         ; DOS-Routine Schreiben

* Jetzt Pufferinhalt schreiben

        move.l   d5,d1             ; Output-Handle nach d1
        move.l   #buffer,d2        ; Textbeginn nach d2
        move.l   d4,d3             ; Textlänge nach d3
        jsr      Write(a6)         ; DOS-Routine Schreiben

* Programmschluß: Lib schließen!

        move.l   a6,a1             ; Basis DOS-Lib
        move.l   4,a6              ; Exec-Basis
        jsr      CloseLib(a6)      ; Lib schließen

ende:   rts                        ; Zum CLI

* Datenbereich: diverse Namen etc.

dosname:        dc.b    "dos.library",0
                even
dosbase:        ds.l    1
text1:          dc.b    "Bitte geben Sie Text ein: "
                even
text2:          dc.b    "Sie haben geschrieben: "
                even
buffer:         ds.b    40
```

Programm 3.2: Texteingabe von der Tastatur

Neu an diesem Programm ist, daß wir die DOS-Basis nicht in einer Variablen sichern, sondern direkt nach a6 schreiben

und zum Schließen von dort nach a1 holen. Das geht, wenn man nur mit einer geöffneten Library arbeitet.

Das Output-Handle sichern wir jetzt im Register d5. Dort ist es genauso sicher wie in einer Variablen. Man sollte nur bei der Wertesicherung nicht zu viel mit Registern herumhantieren, sonst verliert man schnell den Überblick, in welchem Programmteil welche Register verändert werden. In diesem Zusammenhang ist übrigens wichtig, daß die Register a0, a1, d0 und d1 als "Schmierpapier" ("Scratch") gelten und von Library-Routinen verändert werden können. Sie dürfen also nicht erwarten, daß diese vier Register nach einem Aufruf noch die selben Werte enthalten wie vorher. Alle anderen Registerinhalte bleiben allerdings erhalten.

Nach der Ausgabe der Eingabeaufforderung holen wir uns das Input-Handle der Tastatur und rufen die Read-Routine auf. Als Parameter bekommt sie die Startadresse eines 40-Byte-Puffers, den wir im Datenbereich reserviert haben, und eine 40 als Datenlänge. Als Rückgabewert erhalten wir die Anzahl Zeichen, die eingetippt wurden (inklusive des abschließenden Return-Tastendrucks). Diese Zahl merken wir uns in d4:

```
move.l   d0,d1          ; nach d1
move.l   #buffer,d2     ; Start des Eingabe-Puffers
move.l   #40,d3         ; 40 Zeichen max.
jsr      Read(a6)       ; Lesen
move.l   d0,d4          ; Anzahl gelesener Zeichen
                        ; in d4 zwischenspeichern
```

Gewöhnlich sollte man Rückgabewerte von Library-Routinen immer sichern. Hier können wir allerdings das Input-Handle, das wir in d0 bekommen, direkt nach d1 weiterleiten, da wir es ohnehin nur einmal (zum Lesen) brauchen.

Nun geben wir einen zweiten Text aus und direkt dahinter den Inhalt des Eingabepuffers. Die Datenlänge wissen wir noch vom Read-Aufruf (sie entspricht der Anzahl der eingegebenen Zeichen). Der Rest des Programms läuft wie gehabt.

3.3.3 Die Kommandozeile

Die meisten CLI-Befehle brauchen Parameter, die hinter den Befehlsnamen geschrieben werden, z.B.:

```
copy df0:text to df1:
```

Diesen Parametertext hinter dem Befehl nennt man "Kommandozeile". Auch in unseren Assembler-Programmen können wir diese Kommandozeile auswerten. Beim Programmstart steht in a0 ein Zeiger auf den Kommandozeilentext und in d0 seine Länge (einschließlich des Return-Zeichens am Ende).

Da wir schon Text im CLI-Fenster ausgeben können und nun über die Kommandozeile Bescheid wissen, ist es ein Leichtes, den CLI-Befehl "Echo" nachzuvollziehen. Dieser schreibt ja lediglich exakt den Text der Kommandozeile ins Fenster. Folgendes Programm erfüllt die selbe Aufgabe wie Echo:

```
* Programm 3.3: Ausgabe der Kommandozeile (Echo-Befehl)

ExecBase    =       4
OpenLib     =       -552
CloseLib    =       -414
Output      =       -60
Write       =       -48

        movem.l  a0/d0,-(sp)      ; Kommandozeile sichern

        move.l   ExecBase,a6      ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

        jsr      Output(a6)       ; Output-Handle holen
        move.l   d0,d1            ; und nach d1 für Write

        movem.l  (sp)+,a0/d0      ; Kommandozeile zurückholen

        move.l   a0,d2            ; Aufruf Write-Routine
        move.l   d0,d3
        jsr      Write(a6)

        move.l   a6,a1            ; Library wieder schließen
        move.l   ExecBase,a6
        jsr      CloseLib(a6)

        rts                       ; Zum CLI

* Datenbereich

dosname:     dc.b    "dos.library",0
             even
```

Programm 3.3: Ausgabe der Kommandozeile

Bevor wir die DOS-Lib öffnen, müssen wir den Inhalt von a0 und d0 sichern, da diese Register durch Library-Aufrufe verändert werden. Als Sicherungsort wählen wir den Stack, auf den wir die Register mit dem schon besprochenen Befehl MOVEM schreiben:

```
        movem.l  a0/d0,-(sp)      ; Kommandozeile sichern
```

Nach dem Öffnen der Library und dem Abholen des Output-Handles holen wir uns die Kommandozeilen-Register zurück:

```
        movem.l  (sp)+,a0/d0     ; Kommandozeile zurückholen
```

Beim Library-Öffnen verzichten wir diesmal auf den Fehler-Test. Normalerweise sollte man Fehler natürlich abfangen, aber die Chance, daß die DOS-Lib nicht geöffnet werden kann, ist gleich Null.

Nun leiten wir die Kommandozeilen-Register an die von Write geforderten Parameter-Register weiter und rufen Write auf:

```
        move.l   a0,d2           ; Aufruf Write-Routine
        move.l   d0,d3
        jsr      Write(a6)
```

Das war's auch schon. Damit ist unser eigener Echo-Befehl fertig.

3.4 Schleifen in Assembler

Nachdem wir nun Grundlagen zur Textein- und -ausgabe kennengelernt haben, können wir uns mit diversen Programmier-Techniken beschäftigen. Als erstes kommen wir zu den Schleifen-Konstruktionen.

3.4.1 Die REPEAT- und die WHILE-Schleife

Man stelle sich folgendes vor: Ein Puffer von 95 Bytes Größe soll mit den ASCII-Zeichen 32 (Leertaste) bis 126 ("~", genannt "Tilde") gefüllt werden. Dann müßte man, ohne Verwendung einer Schleife, jedes Zeichen einzeln in den Puffer schreiben, also etwa so:

```
        ...
        lea      buffer,a0
        move.b   #32,(a0)+
        move.b   #33,(a0)+
        move.b   #34,(a0)+      usw. usw.
        ...
```

Bild 3.3: Einen Puffer füllen ohne Schleife

Das wäre wohl ein recht aufwendiges Verfahren. Besser ist die Programmierung einer Schleife, deren Zähler von 32 bis 126 läuft und jeweils nach '(a0)+' geschrieben wird.

In Sprachen wie PASCAL oder C gibt es dafür die REPEAT-Schleife (Wiederhole die Anweisung solange bis eine definierte Bedingung zutrifft) und die WHILE-Schleife (Solange die definierte Bedingung zutrifft, wiederhole die Anweisung). Der Unterschied zwischen diesen beiden Typen ist der, daß die Bedingung bei WHILE vor und bei REPEAT nach dem Schleifenkern geprüft wird. Eine REPEAT-Schleife wird also mindestens einmal durchlaufen, eine WHILE-Schleife nicht unbedingt.

Diese Konstruktionen lassen sich natürlich auch in Assembler übertragen. In unserem Beispiel 'Puffer füllen' könnte man schreiben:

```
        ...
        lea     buffer,a0
        move.b  #32,d0
m1:     move.b  d0,(a0)+
        addq    #1,d0
        cmp.b   #127,d0
        blt     m1
        ...
```

Bild 3.4: Puffer füllen per REPEAT-Schleife

Das heißt also: schreibe d0 nach '(a0)+', erhöhe es dann um eins und wiederhole das ganze, solange d0 kleiner als 127 ist. Das wäre die Lösung als REPEAT-Schleife (Prüfung der Bedingung am Ende). Eine WHILE-Lösung ist hier nicht so sinnvoll, da man WHILE nur einsetzt, wenn die Schleife eventuell gar nicht durchlaufen werden darf. Aber zu Demonstrationszwecken wollen wir sie uns trotzdem anschauen:

```
        ...
        lea     buffer,a0
        move.b  #32,d0
m1:     cmp.b   #126,d0
        bgt     m2
        move.b  d0,(a0)+
        addq    #1,d0
        bra     m1
m2:     ...
```

Bild 3.5: Puffer füllen per WHILE-Schleife

Hier wird vor dem Puffer-MOVE geprüft, ob d0 größer ist als 126, und wenn ja, wird nach m2 (hinter der Schleife) verzweigt.

3.4.2 Die DBCC-Schleife

Dieser Schleifentyp entspricht der FOR-Schleife in anderen Sprachen. FOR wiederholt einen Programmteil so oft wie angegeben, ohne daß eine Abbruchsbedingung gestellt wird. In Assembler sind wir da aber sogar etwas nobler, wir können neben der Durchlaufsanzahl auch noch eine Abbruchsbedingung angeben.

Man verwendet dazu den DBCC-Befehl, der eine Sonderform des BCC-Befehl darstellt. Wir erinnern uns: BCC heißt "Branch on Condition Code", also "Verzweige über einen Bedingungscode". DBCC heißt nun "Decrement and Branch on Condition Code", also "Erniedrige und Verzweige über einen Bedingungscode". Erniedrigt wird hier ein Datenregister. Die DBCC-Schleife wird solange wiederholt, bis entweder das Datenregister auf -1 gelaufen ist oder die Bedingung nicht mehr zutrifft, wobei die Bedingungscodes denen von BCC entsprechen. Beispiel:

```
        cmp.b   #10,d0
        dbeq    d1,marke1
```

D1 wird erniedrigt, und es wird solange zu 'marke1' zurückgesprungen, bis d1=-1 ist oder der 'eq'-Vergleich nicht mehr zutrifft, d0 also <> 10 ist. Die DBCC-Vergleiche beziehen sich, genauso wie bei BCC, auf den Zustand der Flags, weshalb auch hier der Vergleichsbefehl unmittelbar vor dem DBCC stehen sollte.

Oft braucht man den Bedingungscode gar nicht, man will also nur eine bestimmte Anzahl von Schleifendurchläufen haben. In diesem Fall schreibt man anstatt DBCC einfach DBRA, das heißt "Decrement and Branch Always" - Erniedrige und Verzweige immer (d.h. natürlich nur, solange das Datenregister noch nicht -1 ist). Beispiel:

```
        dbra    d0,marke2
```

Solange d0 noch nicht -1 ist, wird zu marke2 verzweigt, ohne eine Bedingung zu prüfen. Eine andere Schreibweise für DBRA ist 'DBF':

```
        dbf     d0,marke2
```

Natürlich muß vor dem Schleifenbeginn das Datenregister mit der gewünschten Anzahl von Schleifendurchläufen belegt werden. Genauer gesagt, mit der Anzahl minus eins, da DBCC ja bis -1 läuft. Um einen Programmteil 10 mal durchlaufen zu lassen, schreibt man also folgendes:

```
        move.w  #9,d0
m1:     ...
        ...
        dbra    d0,m1
        ...
```

Das Datenregister wird von DBCC immer nur als Wort angesprochen, weshalb man beim Belegen '.w' verwenden kann. Die maximale Anzahl Schleifendurchläufe beträgt also 65536, für eine größere Anzahl muß man verkettete Schleifen verwenden.

Zum Abschluß noch unser Beispiel mit dem ASCII-Puffer in der DBCC-Version:

```
        lea       buffer,a0
        move.w    #94,d0
        move.b    #32,d1
m1:     move.b    d1,(a0)+
        addq      #1,d1
        dbra      d0,m1
```

Bild 3.6: Puffer füllen mit DBRA

Jetzt brauchen wir zwei Datenregister, eins mit dem derzeitigen ASCII-Wert und eins als Schleifenzähler. Die Schleife soll 95 mal durchlaufen werden, also laden wir d0 mit 94 (DBCC läuft bis -1). Der Rest dürfte eigentlich klar sein.

3.4.3 Zeichen entfernen mit DBCC

Zur Vorbereitung auf das nächste Programm wollen wir uns jetzt noch anschauen, wie man mit einer DBCC-Schleife ein Zeichen aus einer bestehenden Zeichenkette entfernt. Folgendes sei die Aufgabe: Wir haben a3 mit einem Zeiger auf einen 50 Byte langen Puffer geladen, und jetzt soll das 14. Zeichen aus diesem Puffer entfernt (und der Rest dabei aufgerückt) werden.

Die Lösung: Wir kopieren, angefangen beim 15. Zeichen bis zum Ende des Puffers, jeweils das derzeitige Zeichen in das vorhergehende. Das 14. Zeichen wird dabei überschrieben, und das 50., das hinterher doppelt vorhanden ist, überschreiben wir mit 0. Wie oft muß die Schleife dazu durchlaufen werden? Vom 14. bis zum 50. haben wir 37 Zeichen. Da wir aber damit anfangen, das 15. Zeichen ins 14. zu kopieren, müssen wir 36 mal kopieren. Da DBCC bis -1 läuft, kommt eine 35 ins Zähl-Datenregister. Das folgende Programm stellt die Lösung vor:

```
        ...
        lea       buffer,a3
        move.w    #35,d0
        lea       15(a3),a0
        lea       -1(a0),a1
m1:     move.b    (a0)+,(a1)+
        dbra      d0,m1
        clr.b     (a0)
```

Bild 3.7: Zeichen aus Puffer entfernen mit DBRA

Wir belegen das Datenregister a0 mit der Quelle des Kopiervorgangs und a1 mit dem Ziel. Die etwas komisch wirkenden LEA-Befehle sind durchaus sinnvoll: Das 'lea 15(a3),a0' bedeutet: Nehme die Adresse in a3 (der Pufferbeginn), zähle 15 dazu und schreibe die neue Adresse nach a0 (also das Quell-Register). Mit den Befehlen

```
move.l  a3,a0
add.l   #15,a0
```

hätte man natürlich dasselbe erreicht, aber der LEA-Befehl ist schneller und kürzer. Der zweite LEA bewirkt, daß die Zieladresse in a1 genau eins niedriger ist als die Quelladresse, und so soll es ja auch sein. Die Schleife selber ist dann ganz einfach: Sie besteht aus nur einem MOVE-Befehl, der den Inhalt der Adresse in a0 in die Adresse in a1 kopiert (als Byte) und dabei a0 und a1 automatisch um eins erhöht. Der CLR-Befehl nach der Schleife dient zum Löschen des 50., nunmehr doppelt vorhandenen Zeichens.

3.5 Kommandozeile mit Sonderzeichen

Zur Übung und Anwendung der Schleifentechniken wollen wir uns jetzt noch einmal unser Echo-Programm vornehmen. Die CLI-Version dieses Befehls bietet nämlich die Möglichkeit, über die Zeichenkombination '*n' eine neue Zeile zu beginnen und über '*e' ein Escape-Zeichen (Einleitungszeichen für Steuerkommandos wie Fett, Unterstrichen usw.) zu drucken. Wir wollen unser Programm dahingehend verbessern.

Das Verfahren besteht darin, den eingegebenen Text nach Sternchen zu durchsuchen. Haben wir eins gefunden, müssen wir dieses und das darauf folgende Zeichen (n oder e) durch einen Return- bzw. Escape-Code (also nur ein Zeichen) ersetzen. Das heißt, wir müssen auf jeden Fall schonmal ein Zeichen aus dem Text entfernen. Dann wandeln wir das andere Zeichen entsprechend um (der ASCII-Code von Return ist 10 und von Escape 27). Falls das Zeichen nach dem Stern weder n noch e war, lassen wir es, wie es ist. Um einen Stern im Text zu drucken, muß man dann also '**' eingeben.

Hier das Programm:

```
* Programm 3.4: Kommandozeile mit Sonderzeichen

ExecBase    =     4
OpenLib     =     -552
CloseLib    =     -414
Output      =     -60
Write       =     -48
```

```
        movem.l  a0/d0,-(sp)      ; Kommandozeile sichern

        move.l   ExecBase,a6      ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

        jsr      Output(a6)       ; Output-Handle holen
        move.l   d0,d5            ; und in d5 sichern

        movem.l  (sp)+,a4/d4      ; Kommandozeile zurückholen
                                  ; nach a4/d4

* DBCC-Schleife vorbereiten

        move.l   a4,a2            ; Kommandozeile zur Bearbei-
        move.l   d4,d1            ; tung nach a2/d1
        subq     #1,d1            ; Da DBCC bis -1 läuft

* Hauptschleife: Suche nach '*'

lab1:   cmp.b    #"*",(a2)+       ; '*' gefunden?
        beq      lab2             ; Wenn ja, Sprung
        dbra     d1,lab1          ; Schleifen-Ende
        bra      ausg             ; Zur Ausgabe

* Ein Zeichen aus Text entfernen

lab2:   move.l   d1,d0            ; Innerer Schleifenzähler d1
        subq     #1,d0            ; auf d0 minus 1 setzen
        move.l   a2,a0            ; Quell-Textzeiger
        lea      -1(a0),a1        ; Ziel-Textzeiger
lab3:   move.b   (a0)+,(a1)+      ; Verschiebe-Schleife
        dbra     d0,lab3

* Test, ob 'e' oder 'n' auf den Stern folgte

        lea      -1(a2),a0        ; -1 wegen '(a2)+' bei der
                                  ; Suche nach '*'
        cmp.b    #"e",(a0)        ; 'e' gefunden?
        beq      lab4             ; Wenn so
        cmp.b    #"n",(a0)        ; Oder 'n'?
        beq      lab5             ; Dann dahin
        bra      lab6             ; Nichts gefunden

* Ersetzen des 'e' bzw. 'n' durch Esc bzw. Return

lab4:   move.b   #27,(a0)         ; 'e' durch Esc ersetzen
        bra      lab6
lab5:   move.b   #10,(a0)         ; 'n' durch Return ersetzen

* Schleifenzähler und Textlänge anpassen

lab6:   subq     #2,d1            ; Schleifenzähler minus 2
        subq     #1,d4            ; Textlänge minus 1
```

```
        bra       lab1            ; Zur Hauptschleife

ausg:   move.l    d5,d1           ; Ausgabe des Textes
        move.l    a4,d2
        move.l    d4,d3
        jsr       Write(a6)

        move.l    a6,a1           ; Lib schließen und Ende
        move.l    ExecBase,a6
        jsr       CloseLib(a6)

ende:   rts


* Datenbereich

dosname:      dc.b    "dos.library",0
                      even
```

Programm 3.4: Kommandozeile mit Sonderzeichen

Zu diesem Programm ist sicherlich noch einiges zu sagen. Bis zu dem Befehl

```
        movem.l  (sp)+,a4/d4    ; Kommandozeile zurückholen
```

dürfte noch alles klar sein. Mit dem MOVEM-Befehl werden Start und Länge der Kommandozeile vom Stack geholt, allerdings in andere Register als die, von denen sie kamen. Das ist grundsätzlich immer möglich; Registerinhalte, die auf den Stack gelegt wurden, brauchen nicht in genau die selben Register zurückgeschrieben zu werden. Man muß nur auf ihre Reihenfolge achten. Wir schreiben die Kommandozeile hier nach a4 und d4, da wir sie bearbeiten (und damit verändern) müssen, aber auch die Ausgangswerte später noch brauchen.

Als nächstes wird die Hauptschleife - die Suche nach den Sternchen - vorbereitet. Wir holen die Kommandozeile von den "Sicherungsregistern" a4 und d4 in "Arbeitsregister" a2 und d1. A2 zeigt also während der Schleife immer auf das Zeichen der Kommandozeile, das gerade bearbeitet wird, und d1 verwenden wir als Schleifenzähler. Es enthält jeweils die Anzahl der noch zu bearbeitenden Zeichen. Da die DBCC-Schleife bis -1 läuft, müssen wir d1 um eins verkleinern.

Nun folgt die Hauptschleife:

```
lab1:   cmp.b     #"*",(a2)+      ; '*' gefunden?
        beq       lab2            ; Wenn ja, Sprung
        dbra      d1,lab1         ; Schleifen-Ende
        bra       ausg            ; Zur Ausgabe
```

Der CMP-Befehl prüft, ob das aktuelle Zeichen ein Stern ist. Vielleicht verwundert es Sie, daß wir hier einfach "*" schreiben können und nicht den ASCII-Code des Sterns als Zahl verwenden müssen. Bei guten Assemblern ist dies möglich. Bei diesem Test wird das Zeige-Adreßregister auch sofort um eins vergrößert. Wenn ein Stern gefunden wurde, wird nach 'lab2' verzweigt, ansonsten erfolgt der Schleifen-Rücksprung nach 'lab1'. Nach dem Ende der Schleife wird zum Ausgabe-Programmteil verzweigt.

Wenn nun ein Stern gefunden wurde, muß zunächst ein Zeichen aus dem Text entfernt werden, da die Kombinationen '*n' bzw. '*e' (zwei Zeichen) ja in ein Zeichen (Return bzw. Esc) umgewandelt werden. Dazu schreiben wir eine innere Schleife, die in den Zeilen

```
lab2:   move.l  d1,d0           ; Innerer Schleifenzähler d1
        subq    #1,d0           ; auf d0 minus 1 setzen
        move.l  a2,a0           ; Quell-Textzeiger
        lea     -1(a0),a1       ; Ziel-Textzeiger
```

vorbereitet wird. Das Verfahren zum Entfernen eines Zeichens kennen wir ja schon. Unser Quell-Zeichen ist diesmal a2, unser Hauptschleifen-Zeiger (der durch das '(a2)+' im CMP-Befehl ja schon auf dem nächsten Zeichen steht), und das Ziel ist Quellzeichen minus 1.

Der innere Schleifenzähler d0 wird auf 'Hauptschleifenzähler minus 1' gesetzt. Das 'minus 1' ist nötig, weil der Hauptschleifenzähler noch nicht erniedrigt wurde (der DBRA-Befehl wird in der Hauptschleife übersprungen, wenn ein Stern gefunden wird).

Jetzt folgt der Kern der inneren Schleife. Er umfaßt die Zeilen

```
lab3:   move.b  (a0)+,(a1)+     ; Verschiebe-Schleife
        dbra    d0,lab3
```

Also genau das selbe Verfahren wie im vorigen Beispiel-Programm.

Nun haben wir ein Zeichen (nämlich den Stern) entfernt und können fortfahren. Wir müssen jetzt prüfen, ob auf den Stern ein 'n' oder ein 'e' folgte. Dazu holen wir uns mit

```
        lea     -1(a2),a0       ; -1 wegen '(a2)+' bei der
                                ; Suche nach '*'
```

die Adresse des gefragten Zeichens nach a0. Jetzt kommen die Vergleiche:

```
        cmp.b   #"e",(a0)       ; 'e' gefunden?
        beq     lab4            ; Wenn so
        cmp.b   #"n",(a0)       ; Oder 'n'?
```

```
        beq     lab5            ; Dann dahin
        bra     lab6            ; Nichts gefunden
```

Diese Zeilen dürften eigentlich klar sein, ebenso wie die folgenden:

```
lab4:   move.b  #27,(a0)        ; 'e' durch Esc ersetzen
        bra     lab6
lab5:   move.b  #10,(a0)        ; 'n' durch Return ersetzen
```

Zum Abschluß der Sternchen-Bearbeitung müssen noch Hauptschleifenzähler und Textlänge angepaßt werden:

```
lab6:   subq    #2,d1           ; Schleifenzähler minus 2
        subq    #1,d4           ; Textlänge minus 1
```

Die Zeile 'Textlänge minus 1' dürfte klar sein. Der Schleifenzähler muß um zwei erniedrigt werden, weil wir den DBRA-Befehl überspringen, der normalerweise die Reduzierung um eins übernimmt, und außerdem der Text ein Zeichen kürzer geworden ist. Dann folgt der Rücksprung zur Hauptschleife.

Der Rest des Programms (Ausgabe des neuen Textes und Lib schließen) läuft wie bisher.

Damit wäre die Anpassung unseres Echo-Befehls an die CLI-Version abgeschlossen. Im Einleitungstext zum Programm wurde erwähnt, daß das Escape-Zeichen als Einleitungszeichen für spezielle Steuerkommandos gilt. Diese Steuerkommandos sind eine recht interessante Sache, deshalb wollen wir uns nun kurz damit beschäftigen, bevor wir weitergehen.

3.5.1 Die ANSI-Steuerkommandos

ANSI steht für "American National Standard Institute", was etwa dem deutschen DIN entspricht. Tatsächlich stimmen die ANSI-Steuerkommandos auf dem Amiga und den PCs größtenteils überein. Beim Amiga beginnt ein ANSI-Kommando immer mit der Zeichenfolge 'ESC[' (ESC-Taste plus eckige Klammer auf). Das ESC-Zeichen bewirkt, daß die folgenden Zeichen nicht so, wie sie sind, ins Fenster geschrieben werden, sondern daß die ganze Zeichenfolge als Steuerkommando interpretiert wird. Ein interessantes Beispiel ist das Kommando 'ESC[nT'. Es scrollt das Ausgabe-Fenster um n Zeilen nach unten, wobei n als ASCII-Zeichen eingegeben werden muß. Probieren Sie es doch einmal: Geben Sie unserem Echo-Befehl die Kommandozeile

```
*e[10T  (das *e wird durch ein ESC-Zeichen ersetzt)
```

Daraufhin wird das Ausgabefenster um 10 Zeilen nach unten gescrollt. Einige ANSI-Kommandos wollen wir hier zum Experimentieren vorstellen, eine komplette Liste finden Sie im Anhang. Wenn Sie in der folgenden Tabelle einen Kleinbuchstaben finden, der nicht am Ende einer Sequenz

steht, müssen Sie an dieser Stelle eine Zahl, die als ASCII-Zeichen einzugeben ist, einfügen.

ANSI-Sequenz	Wirkung
ESC[z;sH	Setzt den Cursor in Zeile z, Spalte s
ESC[J	Fenster ab Cursorzeile löschen
ESC[K	Zeile ab Cursor löschen
ESC[L	Eine Zeile an der Cursorposition einfügen
ESC[0 p	Cursor ausschalten
ESC[p	Cursor einschalten

Wie gesagt ist dies nur eine Auswahl aus allen verfügbaren ANSI-Sequenzen. Wenn Sie alle kennenlernen wollen, schauen Sie bitte in den Anhang.

3.6 Umrechnung von Zahlensystemen

Bis jetzt können wir die Inhalte der Register oder auch von Speicherstellen nicht direkt als Zahlen auf dem Bildschirm ausgeben, da sie eben als Zahl vorliegen, die Write-Routine aber ASCII-Text verlangt. Beispiel: Wenn in einem Register die Zahl 1 steht, darf nicht einfach eine 1 ausgegeben werden, sondern es muß der ASCII-Code der 1 sein (und der ist 49). Als nächstes wollen wir uns daher mit der Umrechnung von Hex- oder Dezimal-Zahlen in ASCII-Texte und umgekehrt befassen. Zunächst der einfachste Fall:

3.6.1 Umrechnung Hex-Zahl -> ASCII-Text

Eine Hex-Zahl muß auf dem Bildschirm natürlich stellenweise ausgegeben werden, weshalb wir sie auch stellenweise bearbeiten müssen. Mit anderen Worten, wir müssen jeweils eine Stelle der Zahl isolieren und dann umrechnen. Das Umrechnen geschieht einfach durch Aufaddieren des ASCII-Codes von '0' (ASCII-Code 48). Zur Isolierung der Stellen benutzt man im Falle der Hex-Zahl am besten die Bitmanipulations-Befehle. Als Beispiel wollen wir die Zahl $45AB in einen ASCII-Text umwandeln.

Eine Hex-Stelle entspricht bekanntlich 4 Binär-Stellen (auch 1 Nibble genannt). Die einfachste Methode der Isolierung ist also, nur das unterste Nibble der Zahl zu betrachten. Da aber die höchste Hex-Stelle der Zahl zuerst ausgegeben werden soll, müssen wir die die höchste Stelle an die Position der niedrigsten bringen. Das geht am besten mit dem ROL-Befehl. Wir erinnern uns: Der ROL-Befehl schiebt alle Binärstellen einer Zahl nach links, und die links herausfallenden Stellen werden rechts wieder eingespeist. Die Zahl $45AB heißt in binär %0100010110101011. Nach einem Linksrotieren um vier Stellen wird daraus %0101101010110100, die vier höchsten Binärstellen (bzw. die höchste Hex-Stelle) sind also nach "ganz unten" gewandert.

Von der Zahl wollen wir aber zunächst nur das unterste Nibble haben, der Rest wirkt sich recht störend aus. Aus diesem Grund kommt jetzt der logische Befehle AND ins Spiel. Im zweiten Kapitel haben wir erfahren, daß der AND-Befehl die beiden Eingangszahlen Bit für Bit vergleicht und das entsprechende Bit in der Ausgangszahl nur setzt, wenn beide Eingangsbits 1 waren. Mit dem AND-Befehl kann man also problemlos die überflüssigen Stellen "ausblenden". Man sagt auch, die Zahl wird "maskiert". Als Maske dient in unserem Fall die Binärzahl %0000000000001111. Folgende Rechnung wird durchgeührt:

```
Zahl Vorher       % 0101101010110100
Masken-Wert   AND % 0000000000001111
                    ________________
Ergibt            % 0000000000000100
```

Durch die AND-Verknüpfung werden die Bits, die in der Maske auf 0 stehen, also auf jeden Fall 0, die übrigen bleiben unverändert. So haben wir genau die Bits isoliert, die wir haben wollten. Die unteren vier Binärstellen ergeben die Dezimalzahl 4 (was ja auch die erste Stelle unserer Beispielzahl $45AB ist). Auf diese 4 addieren wir nun noch den ASCII-Wert von '0', also 48, und kommen damit auf 52, das dem ASCII-Wert von '4' entspricht.

Dieses Verfahren wiederholen wir für die übrigen drei Hex-Stellen. Das nächste Problem ergibt sich mit der dritten Stelle ($A). $A ist in dezimal 10, wenn wir auf 10 die ASCII-'0' 48 aufaddieren, kommen wir auf 58. Nun ist aber der ASCII-Code für 'A' nicht 58, sondern 65. Wir müssen also den Fall 'Code größer als 57' abfangen, indem wir dann noch eine 7 hinzuaddieren. So kommen wir von der 58 auf die benötigte 65.

Nun das Verfahren als Assembler-Programm. Es erwartet in d0 die umzuwandelnde Zahl und in a0 den Zeiger auf einen Puffer, der groß genug sein sollte (bis zu 8 Zeichen). Die Routine kann Zahlen bis zur maximalen Größe (Langwort, 0 - 4294967296) umwandeln.

```
        moveq    #7,d1          ; Schleife: 8 Nibbles
lab1:   rol.l    #4,d0          ; 1 Nibble durch Rotieren
                                ; ins unterste Nibble holen
        move.b   d0,d2          ; Zur Bearb. umkopieren, da
                                ; Ursprungszahl noch ge-
                                ; braucht wird
        and.b    #$f,d2         ; Zahl maskieren
        add.b    #"0",d2        ; ASCII-Code von '0'
                                ; aufaddieren
        cmp.b    #"9",d2        ; Größer als '9'?
        ble      lab2           ; Wenn nein
        add.b    #7,d2          ; Sonst in 'A'-'F' umrechnen

lab2:   move.b   d2,(a0)+       ; In Puffer schreiben
```

```
        dbra      d1,lab1          ; Schleifenende

        move.b    #0,(a0)          ; Endekennung
```

Bild 3.8: Umrechnung Hex-Zahl in ASCII-Text

Die Hex-Zahl $f in der AND-Zeile entspricht dem binären %0000000000001111, ist allerdings "etwas" kürzer. Der letzte MOVE-Befehl dient dazu, die umgerechnete Zahl im Puffer mit einem Null-Byte (als Endekennung) abzuschließen. Ansonsten dürfte das Programm wohl ziemlich klar sein.

3.6.2 Umrechnung Dezimal-Zahl -> ASCII-Text

So gut das hexadezimale Zahlensystem auch für den Umgang mit dem Computer geeignet ist, manchmal sind dezimale Zahlen doch praktischer, da es unser Gewohnheit entspricht. Zum Beispiel hätte wohl kaum jemand gerne die Größe des freien Speicherplatzes in hex angegeben. Aus diesem Grund schreiben wir nun eine Routine, die eine Zahl in einen Dezimal-ASCII-Text umwandelt.

Der Nachteil des Dezimalsystems ist, daß es nicht so "computernah", also dem Binärsystem verwandt, ist wie das Hexadezimalsystem. Man kann z.B. nicht sagen, so viele Binärstellen entsprechen einer Dezimalstelle (zumindest wäre das keine ganze Zahl). Unsere Methode der Stellen-Isolierung von vorhin kann hier also nicht angewandt werden. Stattdessen führt man eine wiederholte Divison der umzuwandelnden Zahl durch, und zwar eine Division, die ein ganzzahliges Ergebnis und den Rest liefert. Zuerst dividiert man die Zahl durch 10000. Das Ergebnis ist die 10000er-Stelle. Den Divisionsrest (der auf jeden Fall kleiner als 10000 ist) dividiert man durch 1000. Das Ergebnis dieser Division ist die 1000er-Stelle, den Rest verarbeitet man weiter. Dieses Verfahren wird mit 100, 10 und 1 als Dividend durchgeführt. Auf diese Weise kann man alle Stellen der Zahl isolieren.

Bleiben wir beim Beispiel von vorhin. Die Zahl $45AB lautet in dezimal 17835. An ihr wollen wir die Rechnung durchführen (DIV bedeutet ganzzahlige Division):

```
17835 DIV 10000 = 1, Rest 7835
 7835 DIV  1000 = 7, Rest  835
  835 DIV   100 = 8, Rest   35
   35 DIV    10 = 3, Rest    5
    5 DIV     1 = 5, Rest    0
```

Die solchermaßen isolierten Stellen können wir wieder durch Hinzuaddieren der ASCII-'0' in ASCII-Text umrechnen. Dabei brauchen wir diesmal sogar den Sonderfall 'A'-'F' nicht zu berücksichtigen, denn das gibt es ja nur im Hex-System.

Jetzt dürften wir wohl für das Programm gerüstet sein. Es erwartet wieder die Zahl in d0 und den Pufferbeginn-Zeiger in a0.

```
        clr.b    d2              ; Flag führende Nullen
        move.l   #10000,d1       ; Wertigkeit höchste Stelle

lab1:   divu     d1,d0           ; Eine Stelle ausrechnen
        move.b   d0,d3           ; und isolieren
        tst.b    d3              ; Stelle = 0?
        beq      lab2            ; Wenn ja, Sprung

        add.b    #"0",d3         ; in ASCII-Zeichen umrechnen
        move.b   d3,(a0)+        ; Zeichen in Puffer
        moveq    #1,d2           ; Setze Führende-Nullen-Flag
        bra      lab3

lab2:   tst.b    d2              ; Führende Null?
        beq      lab3            ; Wenn ja, weglassen
        move.b   #"0",(a0)+      ; Keine führende Null

lab3:   and.l    #$ffff0000,d0   ; Divisons-Rest isolieren
        swap     d0              ; und ins untere Wort
        cmp.w    #1,d1           ; Letzte Stelle bearbeitet?
        beq      lab4            ; Wenn ja, Sprung

        divu     #10,d1          ; Neue Stellenwertigkeit
        bra      lab1            ; Rücksprung

lab4:   move.b   #0,(a0)         ; Endekennung
```

Bild 3.9: Umrechnung Dezimal-Zahl in ASCII-Text

In d1 halten wir die Zahl, durch die jeweils dividiert wird. Beim ersten Durchlauf ist das 10000, beim zweiten 1000 usw. Die Umrechnung ist abgeschlossen, wenn die letzte Stelle bearbeitet wurde (d1 beim Durchlauf also 1 war).

Erklärt werden muß noch die Sache mit dem "Führende-Nullen-Flag". Zu Beginn der Routine wird das Register d2 gelöscht. Sobald die erste Stelle ungleich Null in den Puffer kommt, wird es gesetzt (auf 1). Solange es also nicht gesetzt ist, sind alle Nullstellen, führende Nullstellen (Nullen vor der Zahl). Solche Nullstellen werden nicht mit in den Puffer geschrieben. Sobald die erste Nicht-Null-Stelle erscheint, müssen natürlich auch alle nachfolgenden Nullen in den Puffer. Dazu dient das Register d2.

Noch ein Wort zum Divisionsrest: Der DIV-Befehl schreibt ja das Ergebnis ins untere und den Rest ins obere Wort eines Langwort-Registers. Das Ergebnis kann nach der Disision einfach mit einem MOVE.B-Befehl herausgeholt werden, da es ja sowieso nie größer als 9 sein kann:

```
        move.b   d0,d3           ; und isolieren
```

Um nun an den Rest heranzukommen, wird zunächst das untere Wort durch AND-Maskierung mit $FFFF0000 ausgeblendet, dann wird das obere Wort mit dem unteren per SWAP-Befehl vertauscht:

```
lab3:   and.l   #$ffff0000,d0  ; Divisons-Rest isolieren
        swap    d0             ; und ins untere Wort
```

Nach der Verkleinerung der Stellenwertigkeit mit

```
        divu    #10,d1         ; Neue Stellenwertigkeit
```

erfolgt der Rücksprung zur Hauptschleife.

3.6.3 Umrechnung Dezimal-Langwort -> ASCII-Text

So weit, so gut. Die Routine hat nur einen entscheidenden Nachteil: Sie kann nur Zahlen von 0 bis 65535 (also Wort-Größe) bearbeiten, da der DIVU-Befehl das Ergebnis und den Rest der Division jeweils in ein Wort packen muß. Wenn wir ein Langwort umrechnen wollen, müssen wir uns etwas anderes überlegen.

Das Verfahren kann im Prinzip beibehalten werden, nur die Division müssen wir anders realisieren. Schauen wir uns doch mal an, was Multiplikation und Division überhaupt bedeuten:

Anstatt '3 * 5' kann ich auch '5 + 5 + 5' schreiben. Ebenso gibt das Ergebnis von '20 / 4' an, wie oft ich 4 aufaddieren muß, um auf 20 zu kommen. Daraus folgt, daß wir, anstatt unsere umzurechnende Zahl durch die Stellenwertigkeiten zu dividieren, auch die Stellenwertigkeit solange von ihr abziehen können, bis es einen Unterlauf gibt. Somit wird die Stelle auch isoliert. Als Beispiel nehmen wir uns die Zahl 13241 vor:

```
1. Subtr.: 13241 - 10000 = 3241
2. Subtr.:  3241 - 10000 = -6758 (Unterlauf)
```

Die Subtraktion war einmal möglich, also ist die erste Stelle eine 1. Mit dem Rest (dem letzten vor dem Unterlauf) wird weitergerechnet:

```
1. Subtr.:  3241 -  1000 = 2241
2. Subtr.:  2241 -  1000 = 1241
3. Subtr.:  1241 -  1000 =  241
4. Subtr.:   241 -  1000 = -759 (Unterlauf)
```

Da 3 Subtraktionen ohne Unterlauf möglich waren, ist die zweite Stelle eine 3. Jetzt dürfte das System wohl klar sein. Das Auftreten eines Unterlaufs bei der Subtraktion überwachen wir über das C-Flag (Carry), das bei Über- und Unterläufen gesetzt wird. Schauen wir uns jetzt das zugehörige Programm an (wie immer Zahl in d0 und Pufferzeiger in a0):

```
        clr.b   d2              ; Führende-Nullen-Flag
        lea     values,a1       ; Tabelle der Wertigkeiten

lab1:   clr.b   d1              ; Zähler für gelungene
                                ; Subtraktionen
lab2:   addq    #1,d1           ; Subtr.-Zähler erhöhen
        sub.l   (a1),d0         ; Subtraktion durchführen
        bcc     lab2            ; Wenn kein Übertrag,
                                ; nochmal subtrahieren
        add.l   (a1),d0         ; Letzte Subtr. und Zähler-
        subq    #1,d1           ; erhöhung rückgängig

        tst.b   d1              ; Stelle = 0?
        beq     lab3            ; Wenn ja, Sprung

        add.b   #"0",d1         ; In ASCII umrechnen
        move.b  d1,(a0)+        ; In den Puffer
        moveq   #1,d2           ; Führende-Nullen-Flag
        bra     lab4

lab3:   tst.b   d2              ; Führende Null?
        beq     lab4            ; Wenn ja
        move.b  #"0",(a0)+      ; Sonst Null in Puffer

lab4:   cmp.l   #1,(a1)         ; Letzte Stelle bearbeitet?
        beq     lab5            ; Wenn ja

        add.l   #4,a1           ; Nächste Wertigkeit
        bra     lab1            ; Zur Hauptschleife

lab5:   move.b  #0,(a0)         ; Endekennung

        ...

values: dc.l    1000000000,100000000,10000000
        dc.l    1000000,100000,10000,1000,100,10,1
```

Bild 3.10: Umrechnung Dezimal-Langwort in ASCII-Text

Das Flag für die führenden Nullen arbeitet genauso wie im letzten Programm. Zur Stellenwertigkeits-Bestimmung ist folgendes zu sagen: Wir können jetzt nicht mehr einfach mit der größten Wertigkeit anfangen und diese dann jeweils für die nächste Stelle durch 10 teilen, da die größte Wertigkeit 1000000000 ist, was die Wort-Beschränkung von DIVU eindeutig überschreitet. Wir verwenden daher eine Tabelle, in der die Wertigkeiten der einzelnen Stellen aufgeführt sind. Zu Beginn des Programms holen wir uns den Beginn dieser Tabelle nach a1:

```
        lea     values,a1       ; Tabelle der Wertigkeiten
```

Auf die Wertigkeiten wird dann per ARI zugegriffen:

```
        sub.l    (a1),d0          ; Subtraktion durchführen
```

Für die nächste Stelle wird einfach der Tabellenzeiger a1 um 4 erhöht (da jede Wertigkeit ein Langwort, also vier Bytes, belegt):

```
        add.l    #4,a1            ; Nächste Wertigkeit
```

3.6.4 Umrechnung ASCII-Text -> Hex-Zahl

Jetzt können wir also Zahlen, die irgendwo in unseren Programmen vorkommen, in "Klartext" umwandeln. Was nützlich wäre, sind Umwandlungsroutinen von Klartext-Zahlen (etwa aus CLI-Eingaben) in computerinterne Zahlen. Dazu kommen wir jetzt.

Als erstes die Interpretation eines ASCII-Textes als Hex-Zahl. Dazu geht man quasi den umgekehrten Weg wie bei der Umrechnung Hex-Zahl -> ASCII-Text. Man nimmt sich Stelle für Stelle vor, rechnet den ASCII-Wert in eine Zahl um und schiebt die Stellen von rechts nach links in die Ergebniszahl hinein. Am besten schauen wir uns zuerst das Programm an, das erleichtert die Erklärungen. Es erwartet in a0 den Zeiger auf den Puffer, in dem die ASCII-Zahl steht. Als Endekennzeichen wird ein Null-Byte erwartet. In d0 steht hinterher die umgerechnete Zahl:

```
        clr.l    d0               ; Ergebnis löschen
lab1:   move.b   (a0)+,d1         ; Eine Stelle bearbeiten

        cmp.b    #58,d1           ; Stelle 'A'-'F'?
        blt      lab2             ; Wenn nein
        bclr     #5,d1            ; Force Uppercase
        sub.b    #7,d1            ; Auf 10-15 umrechnen

lab2:   sub.b    #48,d1           ; ASCII-'0' abziehen
        or.b     d0,d1            ; Mit oberem Nibble des
        move.b   d1,d0            ; Ergebnisbytes verknüpfen

        tst.b    (a0)             ; Noch weitere Stellen?
        beq      lab3             ; Wenn nein

        rol.l    #4,d0            ; Ergebniszahl rotieren
        bra      lab1             ; Rücksprung

lab3:   ...
```

Bild 3.11: Umrechnung ASCII-Text in Hex-Zahl

Es wird jeweils eine Stelle zur Bearbeitung nach d1 geholt. Dabei wird a0 automatisch erhöht. Wenn die Hex-Stelle 'A'-'F' war, muß zunächst mal ein Großbuchstabe "erzwungen" wer-

den (Force Uppercase). Das geschieht, indem das 5. Bit im ASCII-Code gelöscht wird:

```
        bclr    #5,d1           ; Force Uppercase
```

Falls es schon vorher ein Großbuchstabe war, wurde dieses Bit sowieso gelöscht, und es ändert sich nichts. Im Falle von Kleinbuchstaben entspricht das Bitlöschen einer Subtraktion von 32, da die Wertigkeit der 5. Binärstelle (von 0 ab gezählt) 32 ist. Die ASCII-Codes der Kleinbuchstaben sind alle um 32 größer als die der entsprechenden Großbuchstaben, daher die Subtraktion. Dann wird vom ASCII-Code 7 abgezogen, damit er sich direkt an die Codes für '0'-'9' anschließt.

Nun wird die ASCII-Zahl durch Abziehen von 48 (ASCII-'0') in eine "normale" Zahl umgerechnet. Diese Zahl muß quasi ganz rechts in die Ergebniszahl "eingeschoben" werden, wobei die schon vorhandenen Stellen nach links rutschen. Nach dem Einschieben der letzten ASCII-Stelle steht dann die erste ASCII-Stelle automatisch da, wo sie hin muß, nämlich an der höchsten Hex-Stelle.

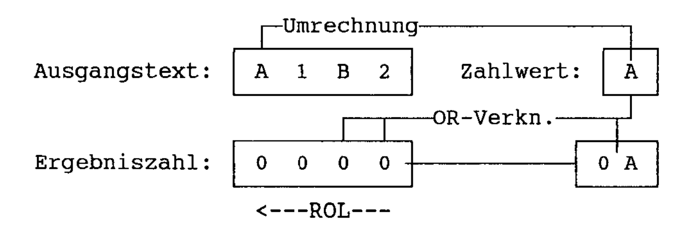
Das "Einschieben" geschieht, indem die Stellenzahl, die höchstens 15 beträgt (also ein Nibble groß ist), zuerst mit der schon vorhandenen Ergebniszahl OR-verknüpft wird und das Verknüpfungsergebnis als neue Ergebniszahl verwendet wird:

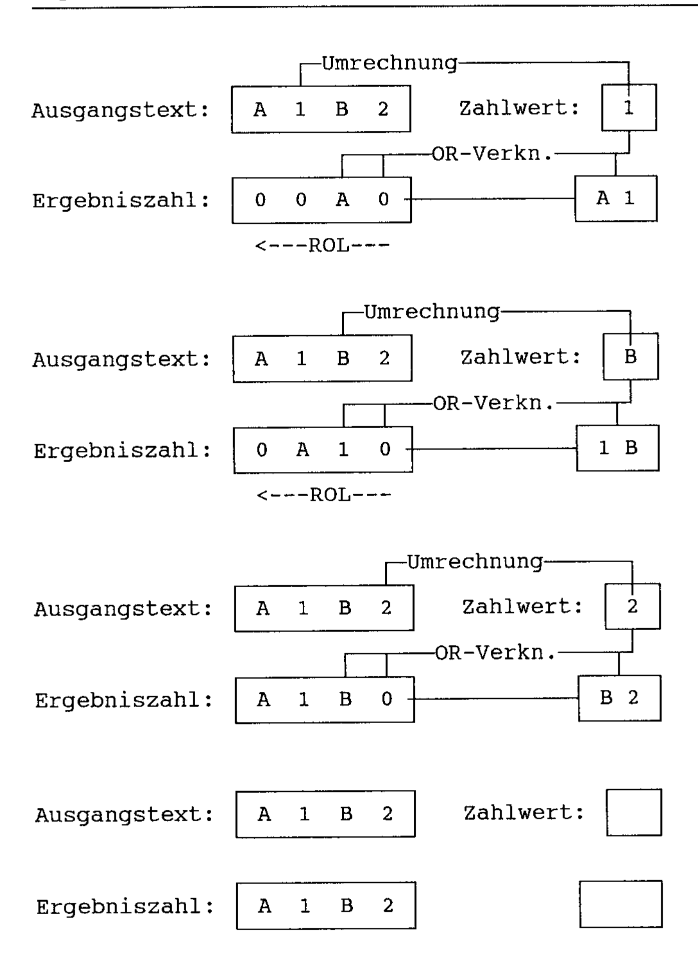
```
        or.b    d0,d1           ; Mit oberem Nibble des
        move.b  d1,d0           ; Ergebnisbytes verknüpfen
```

Wir erinnern uns: Beim Umrechnen Hex-ASCII mußten wir eine AND-Verknüpfung durchführen, um das unterste Nibble der Zahl zu isolieren. Jetzt gehen wir den umgekehrten Weg: Wir verbinden das neue Ergebnisnibble mit der schon vorhandenen Zahl. Dann wird die Ergebniszahl mittels

```
        rol.l   #4,d0           ; Ergebniszahl rotieren
```

um vier Binärstellen (also um eine Hex-Stelle) nach links geschoben (falls noch weitere Stellen folgen), damit das nächste Nibble unten Platz hat. Die folgende Grafik soll den Ablauf noch etwas deutlicher machen:





Die OR-Verknüpfung ist notwendig, da wir die neuen Nibbles ja in der Ergebniszahl einschieben müssen, ohne andere Nibbles in dieser zu überschreiben. Die kleinste "Ansprechgröße" für ein Register ist das Byte. Da wir ein halbes Byte hinzufügen wollen, aber nur byteweise zugreifen können, müssen wir die Verknüpfung der beiden Nibbles des untersten Bytes auf einer noch tieferen Ebene, eben auf Bit-Ebene mittels OR, durchführen.

3.6.5 Umrechnung ASCII-Text -> Dezimal-Zahl

Die Interpretation eines Textes als Dezimal-Zahl ist wieder etwas einfacher. Wir gehen auch hier den umgekehrten Weg wie bei der Umrechnung Zahl-Text. Jede ASCII-Stelle wird in eine

Zahl umgerechnet. Dann wird so oft, wie die Zahl angibt, die Stellenwertigkeit auf eine Hilfszahl (die vor jedem Stellendurchlauf gelöscht wird) aufaddiert, und diese dann auf die Ergebniszahl. Hier das Programm (Pufferzeiger in a0, Text mit Nullbyte abgeschlossen, Ergebnis hinterher in d0):

```
        move.l  a0,a1           ; String-Beginn kopieren
lab1:   tst.b   (a1)+           ; String-Ende erreicht?
        bne     lab1            ; Wenn nein
        sub.l   #1,a1           ; Wegen '(a1)+'

        clr.l   d0              ; Ergebnis löschen
        lea     values,a2       ; Wertigkeitstabelle

lab2:   clr.l   d2              ; Zwischenspeicher löschen
        clr.l   d3
        move.b  -(a1),d2        ; Eine Stelle isolieren
        sub.b   #"0",d2         ; ASCII-'0' abziehen
        tst.b   d2              ; Stelle = 0?
        beq     lab4            ; Wenn ja
        subq    #1,d2           ; DBRA läuft bis -1

lab3:   add.l   (a2),d3         ; Wertigkeit aufaddieren
        dbra    d2,lab3         ; Schleifenende

        add.l   d3,d0           ; Auf Ergebnis aufaddieren

lab4:   cmp.l   a0,a1           ; Höchste Stelle erreicht?
        beq     lab5            ; Wenn ja

        add.l   #4,a2           ; Stellenwertigkeit erhöhen
        bra     lab2            ; Rücksprung

lab5:   ...

values: dc.l    1,10,100,1000,10000,100000,1000000
        dc.l    10000000,100000000,1000000000
```

Bild 3.12: Umrechnung ASCII-Text in Dezimal-Zahl

Bei diesem Verfahren müssen wir die ASCII-Zahl "von hinten aufrollen", also bei der Stelle mit der niedrigsten Wertigkeit anfangen. Das ist nötig, da wir nicht wissen, wieviele Stellen die Zahl überhaupt hat (führende Nullen sind nicht zwingend), mit welcher Wertigkeit wir also anfangen müßten.

Im ersten Teil des Programms wird daher das Abschluß-Nullbyte gesucht und der Zahltext dann von da aus zu den fallenden Adressen hin bearbeitet (die höchste Stelle steht ja weiter vorne im Speicher).

Noch ein paar Worte zu den beiden CLR-Befehlen bei 'lab2'. Der erste CLR-Befehl ist nötig, da wir ein paar Zeilen später d2 über einen 'move.b'-Befehl belegen. Es wird also nur

das unterste Byte des Registers verändert, alles andere, was eventuell schon darin enthalten ist, bleibt bestehen. Da wir aber nur den Byte-Wert, den wir hineinschreiben, im Register haben wollen, müssen wir es zuvor mit 'clr.l' komplett löschen. Der zweite CLR-Befehl muß sein, da wir auf d3 nur Zahlen aufaddieren, aber nichts direkt hineinschreiben. Ein eventuell schon vorhandener Inhalt muß also vor der ersten Addition gelöscht werden.

Damit hätten wir nun die wichtigsten Umrechnungsroutinen beisammen. Denken Sie nicht, daß wir sie nur so aus Spaß besprochen haben. Im Abschluß-Programm dieses Kapitels, das ein kleiner Taschenrechner-Befehl für den CLI sein wird, kommen sie alle zum Einsatz, ebenso wie die Techniken, die wir in den nächsten Abschnitten lernen werden.

3.7 Unterprogramme

Wie geht nun der Einbau der Umrechnungsroutinen in unsere Programme vonstatten? Ganz einfach, fügen Sie die Routine in ihren Quelltext ein, aber nicht überall dort, wor sie diese brauchen. Die Routine wird mit bestimmten Befehlen versehen, damit sie als Unterprogramm aufgerufen werden kann und kommt dann nur einmal in den Quelltext. Folgende Punkte müssen erfüllt sein, damit eine Routine als Unterprogramm dienen kann:

1. RTS am Ende

Die Routine wird mit einem BSR-Befehl (Branch to Subroutine) aufgerufen. Damit ins Hauptprogramm zurückgekehrt wird muß Sie mit RTS (Return from Subroutine) enden.

2. Register retten

Es ist üblich, die Register a0, a1, d0 und d1 als "Schmierpapier" ("Scratch") anzusehen. Das heißt, jedes Unterprogramm (und insbesondere jede Library-Routine) kann sie verändern. Die übrigen Register aber müssen erhalten bleiben. Wenn Sie also andere Register als die Scratch-Register verwenden, müssen Sie diese vor der ersten Veränderung sichern, am besten mit MOVEM auf dem Stack, und vor dem Routinenende wieder zurückholen.

3. Parameterübergabe vereinbaren

Die Parameter, die die Routine erwartet, müssen in fest vereinbarten Registern oder Adressen o.ä. stehen. Für jede Library-Routine ist z.B. genau festgelegt, welche Parameter in welche Register müssen. Rückgabewerte kommen gewöhnlich ins Register d0, aber das können Sie auch variieren.

4. Nur routinen-interne Referenzen

Das bedeutet, daß Sie innerhalb einer Unterroutine nur Labels u.ä. verwenden dürfen, die in der Routine selber vorkommen. Die Routine darf nicht an ein bestimmtes übergeordnetes Programm gebunden sein.

Wir haben schon des öfteren Unterprogramme benutzt, nämlich die Library-Routinen (diese werden ja über JSR aufgerufen). Eigene Unterprogramme geschrieben haben wir jedoch noch nicht. Das wollen wir jetzt nachholen. Als erstes Beispiel schauen wir uns einmal die Hex-ASCII-Konvertierungsroutine als Unterprogramm an:

```
hexascii:
        movem.l  d1/d2,-(sp)     ; Register retten

        moveq    #7,d1           ; Schleife: 8 Nibbles
lab1:   rol.l    #4,d0           ; 1 Nibble durch Rotieren
                                 ; ins unterste Nibble holen
        move.b   d0,d2           ; Zur Bearb. umkopieren, da
                                 ; Ursprungszahl noch ge-
                                 ; braucht wird
        and.b    #$f,d2          ; Zahl maskieren
        add.b    #"0",d2         ; ASCII-Code von '0'
                                 ; aufaddieren
        cmp.b    #"9",d2         ; Größer als '9'?
        ble      lab2            ; Wenn nein
        add.b    #7,d2           ; Sonst in 'A'-'F' umrechnen

lab2:   move.b   d2,(a0)+        ; In Puffer schreiben
        dbra     d1,lab1         ; Schleifenende

        move.b   #0,(a0)         ; Endekennung

        movem.l  (sp)+,d1/d2     ; Register zurückholen
        rts                      ; Zurück zum Hauptprogramm
```

Bild 3.13: Die Hex-ASCII-Routine als Unterprogramm

So sehr viel hat sich gar nicht verändert. Zwei MOVEM-Befehle, ein RTS und ein Label sind hinzugekommen. Die Labels, die benutzt werden, sind alle routinen-intern, und auch die Parameter sind festgelegt: d0 enthält die Zahl und a0 den Zeiger auf den Ausgabe-Puffer. Wichtig ist das neue Label: Es stellt die Einsprungstelle der Subroutine dar.

Ein Aufruf dieser Routine könnte so aussehen:

```
        move.l   #$AB12CD34,d0   ; Testzahl
        lea      buffer,a0       ; Puffer für ASCII-Zahl
        bsr      hexascii        ; Aufruf der Routine
        ...                      ; Verarbeitung des Puffers
```

Das wars vorläufig auch schon zum Thema Unterprogramme. Im Abschlußprogramm dieses Kapitels werden wir wieder Unterprogramme einsetzen. Jetzt kommen wir zuerst mal zu einem anderen Thema.

3.8 Mehrfachverzweigungen

In vielen Programmen kommen diverse Auswahlmenüs vor. Wenn der Benutzer einen Menüpunkt gewählt hat, muß das Programm entsprechend dieser Wahl zu bestimmten Teilen (meist Unterprogrammen) verzweigen. Eine Möglichkeit, diese Mehrfach-Verzweigung zu realisieren, ist die Benutzung einer IF-Kette:

3.8.1 Mehrfachverzweigung mit einer IF-Kette

Angenommen, der Benutzer wird in einem Menü aufgefordert, die gewünschte Programmfunktion durch Eingabe von '1', '2' oder '3' usw. auszuwählen. Das eingegebene Zeichen soll nach d0 geschrieben werden. Eine Verzweigung könnte dann so aussehen:

```
cmp.b   #"1",d0
beq     prg1
cmp.b   #"2",d0
beq     prg2
cmp.b   #"3",d0
beq     prg3
...
```

Diese Methode hat aber zwei Nachteile: Erstens artet sie bei vielen Menüpunkten in eine Menge Schreibarbeit aus (von der Speicherplatzverschwendung mal ganz abgesehen), und außerdem dürfen die einzelnen Programmteile höchstens 32 KByte von der Verzweigekette entfernt sein (da die Zieladresse von BCC auf +32767 bis -32768 beschränkt ist), was bei größeren Programmen zu Schwierigkeiten führen kann.

Solange Sie nur kleine Programme mit wenigen Menüpunkten (bis zu 8 Stück) schreiben, können Sie eine IF-Kette zur Verzweigung benutzen, ansonsten sollten Sie eine der folgenden Methoden verwenden.

3.8.2 Mehrfachverzweigung mit Sprungtabelle

Diese Verzweigungstechnik basiert auf einer Tabelle, in der die Einsprungadressen aller Routinen verzeichnet sind. Aus dem eingegebenen Zeichen wird dann die Tabellenplatznummer des gewählten Menüpunktes berechnet.

Das Anlegen dieser Tabelle ist ganz einfach. Angenommen, Sie haben folgende Routinen in Ihrem Programm:

```
prg1:   ...                     ; Beginn der ersten Routine
        ...
        rts
prg2:   ...                     ; Beginn der zweiten Routine
        ...
        rts
prg3:   ...                     ; Beginn der dritten Routine
        ...
        rts                     ; usw. usw.
```

Dann legen Sie die Sprungadressen-Tabelle so an:

```
table:        dc.l    prg1,prg2,prg3
```

Mit der DC-Direktiven kann man beliebige Werte ins Programm einfügen, also auch Adressen. Labels repräsentieren die Adressen der Befehle oder Daten, vor denen sie stehen. Die Labels vor der ersten Zeile einer Routine repräsentieren dann natürlich die Startadresse der Routine, weshalb wir sie zum Aufbau der Tabelle benutzen können. Wichtig ist, daß die Einträge der Tabelle in der Reihenfolge der zugehörigen ASCII-Zeichen stehen müssen. In unserem Beispiel muß also zuerst die Routine, die bei ASCII-'1' aufgerufen wird, kommen, dann die Routine für ASCII-'2' usw.

Wenn in d0 das eingegebene Zeichen ('1', '2' oder '3') abgelegt wird, können wir aus ihm folgendermaßen die Startadresse der anzuspringenden Routine ableiten:

```
        lea      table,a0        ; Tabellenstart
        sub.b    #"1",d0         ; ASCII in Zahl umrechnen
        asl      #2,d0           ; d0 mal 4
        move.l   0(a0,d0),a1     ; Startadresse aus Tabelle
        jsr      (a1)            ; Routine anspringen
```

Bild 3.14: Einsprungadresse berechnen (1 Tabelle)

Der Start der Tabelle wird nach a0 geholt. Das Zeichen in d0 wird durch Subtraktion des ASCII-Codes des kleinstmöglichen Auswahlzeichens (hier also die '1') in eine Zahl (0-2) umgerechnet. Diese Zahl gibt die Platznummer der gesuchten Einsprungadresse in der Tabelle an. Da jeder Tabelleneintrag 4 Bytes lang ist (Adresse=Langwort), wird die Zahl noch mit 4 malgenommen. An der Speicherstelle 'Tabellenstart plus Zahl' (Adressierungsart ARI mit Index und Offset) ist dann die gewünschte Einsprungadresse zu finden. Diese wird nach a1 geholt und angesprungen (JSR-Befehl indirekt über Adreßregister), das wars.

Zu diesem Thema wollen wir uns auch ein kleines Komplett-Beispielprogramm anschauen. Es soll ein Zeichen der Kommandozeile auswerten und über eine Sprungtabelle einige Routinen anspringen. Als Eingabe-Zeichen lassen wir '1', '2' und

'3' zu, und die einzelnen Routinen sollen eine kleine Meldung ausgeben.

```
* Programm 3.5: Mehrfachverzweigung mit Sprungtabelle

ExecBase     =       4
OpenLib      =       -552
CloseLib     =       -414
Write        =       -48
Output       =       -60

        movem.l  a0,-(sp)         ; Kommandozeile sichern

        move.l   ExecBase,a6      ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

        jsr      Output(a6)       ; Output-Handle holen
        move.l   d0,d4

        movem.l  (sp)+,a0         ; Kommandozeile zurückholen
        clr.l    d0

* Einsprungadresse berechnen

        move.b   (a0),d0          ; Eingebenes Zeichen nach d0
        lea      table,a0         ; Tabellenanfang
        sub.b    #"1",d0          ; ASCII -> Zahl
        asl      #2,d0            ; d0 mal 4
        move.l   0(a0,d0),a1      ; Einsprungadresse nach a1
        jsr      (a1)             ; Routine anspringen

        move.l   a6,a1            ; Lib schließen
        move.l   4,a6
        jsr      CloseLib(a6)

        rts                       ; Zum CLI

ausg1:  move.l   d4,d1            ; Text 1 ausgeben
        move.l   #text1,d2
        move.l   #17,d3
        jsr      Write(a6)
        rts

ausg2:  move.l   d4,d1            ; Text 2 ausgeben
        move.l   #text2,d2
        move.l   #17,d3
        jsr      Write(a6)
        rts

ausg3:  move.l   d4,d1            ; Text ausgeben
```

```
        move.l  #text3,d2
        move.l  #17,d3
        jsr     Write(a6)
        rts

* Datenbereich

dosname:        dc.b    "dos.library",0
                even

table:          dc.l    ausg1,ausg2,ausg3

text1:          dc.b    "Kommandozeile: 1",10
                even
text2:          dc.b    "Kommandozeile: 2",10
                even
text3:          dc.b    "Kommandozeile: 3",10
                even
```

Programm 3.5: Mehrfachverzweigung mit Sprungtabelle

Die Funktionsweise dieses Programms dürfte eigentlich klar sein. Anzumerken ist noch, daß wir hier nur die Adresse der Kommandozeile (a0) auf dem Stack sichern. Die Länge brauchen wir nicht, da wir sowieso nur ein Zeichen auswerten wollen. Sie sollten beachten, daß das Programm den Fall, daß sie ein anderes Zeichen als '1', '2' oder '3' (oder auch gar kein Zeichen) in die Kommandozeile schreiben, nicht abfängt. Das Programm im nächsten Abschnitt wird dies aber tun.

Einen Nachteil hat diese Verzweigungs-Methode noch: Die Menüpunkt-ASCII-Codes müssen alle hintereinander stehen, da aus ihnen direkt der Tabellenplatz bestimmt wird. Sie können also nur Menüs der Art "1=Laden, 2=Speichern, 3=Drucken, 4=Ende usw." benutzen, aber nicht z.B. "L=Laden, S=Speichern, D=Drucken, E=Ende". Zur Realisierung solcher Abfragen kommen wir jetzt.

3.8.3 Mehrfachverzweigung mit zwei Tabellen

Die Benutzung von zwei Tabellen läßt auch nicht-aufeinanderfolgende Menüpunkt-ASCII-Codes zu. In der ersten Tabelle stehen die möglichen ASCII-Zeichen und in der zweiten die Einsprungadressen, und zwar in der selben Reihenfolge wie die zugehörigen Menüzeichen. Bleiben wir beim Beispiel mit dem "Laden, Speichern, Drucken, Ende". Die Routinen könnten dann folgendermaßen im Programm stehen:

```
load:   ...                     ; Start der Lade-Routine
        rts
save:   ...                     ; Start der Speicher-Routine
        rts
print:  ...                     ; ...
```

```
        rts
quit:   ...                     ; ...
```

Die erste Tabelle enthält die zugelassenen Buchstaben:

```
table1:     dc.b    "L","S","D","E"
table1end:
```

Das zweite Label unter der Tabelle dient als "Endemarke". Es repräsentiert die erste Adresse direkt nach der Tabelle. Wenn wir später beim Suchen des Zeichens an diese Adresse kommen, wissen wir, daß die Tabelle zuende ist (Zeichen wurde nicht gefunden). In der zweiten Tabelle stehen wieder die Einsprungadressen:

```
table2:     dc.l    load,save,print,quit
```

Die Berechnung des Tabellenplatzes und damit der Einsprungadresse läuft dann so ab (in d0 soll das ausgewählte Zeichen stehen):

```
        lea     table1,a0       ; Start der Zeichen-Tabelle
        lea     table1end,a1    ; Ende der Zeichen-Tabelle
        bclr    #5,d0           ; Force Uppercase
        clr.l   d1              ; Zähler löschen
lab1:   cmp.b   (a0)+,d0        ; Zeichen gefunden?
        beq     lab2            ; Wenn ja
        addq    #1,d1           ; Sonst Zähler erhöhen
        cmp.l   a1,a0           ; Tabellenende erreicht?
        bne     lab1            ; Wenn nein
        bra     notfound        ; Zeichen nicht gefunden
lab2:   asl     #2,d1           ; Zähler mal 4
        lea     table2,a0       ; Start Adreß-Tabelle
        move.l  0(a0,d1),a1     ; Einsprungadresse holen
        jsr     (a1)            ; Anspringen
```

Bild 3.15: Berechnung der Einsprungadresse (2 Tabellen)

Dieses Verfahren ist ein wenig aufwendiger, bietet aber große Vorteile. Als erstes muß das in d0 stehende Zeichen in der ersten Tabelle gesucht werden. Dazu holen wir den Start der Tabelle nach a0 und ihr Ende (bzw. Ende+1) nach a1. Das Zeichen in d0 wandeln wir in einen Großbuchstaben um. Wir müssen zählen, an der wievielten Stelle in der Tabelle das Zeichen gefunden wurde, also ernennen wir d1 zum Zähler (der zunächst gelöscht wird). Dann vergleichen wir immer ein Zeichen aus der Tabelle mit dem zu suchenden Zeichen, wobei der Tabellenzähler automatisch erhöht wird. Haben wir das Zeichen gefunden, wird nach 'lab2' verzweigt. Ansonsten wird der Tabellenzähler erhöht und zurückgesprungen, falls das Tabellenende noch nicht erreicht wurde (ansonsten springen wir nach 'notfound', wo entsprechend reagiert werden muß).

Bei 'lab2' wird der Tabellenzähler, der ja jetzt die Platznummer des gesuchten Zeichens in der ersten Tabelle und damit auch die der gesuchten Adresse in der zweiten angibt, mit vier malgenommen. Die Adresse wird aus der Tabelle gelesen und angesprungen. Damit hat sich die Sache.
Schauen wir uns noch ein Komplett-Beispielprogramm an, dann wird es bestimmt klarer. Es soll wieder die Kommandozeile auswerten und bei eingegebenem 'L', 'S', 'D' oder 'E' eine kleine Meldung ausgeben. Diesmal wird auch der Fall 'nicht gefunden' berücksichtigt.

Im letzten Programm mußten wir für die Textausgaben drei fast identische Programmteile schreiben. Diesmal verwenden wir dafür ein Unterprogramm (genannt 'print'), dem nur die Startadresse des Textes übergeben wird. Das Output-Handle wird in d4 vorausgesetzt, und die Länge des Textes steht als Byte-Wert im ersten Zeichen des Textes, das dann natürlich nicht mit ausgegeben wird. Einen solchen Text, der als erstes Zeichen seine Länge enthält, nennt man übrigens "BCPL-String". BCPL ist eine C-ähnliche Programmiersprache, aber auch in Pascal werden beispielsweise die Strings (Zeichenketten) so verwaltet. Jetzt aber das Programm:

```
* Programm 3.6: Mehrfachverzweigung mit zwei Tabellen

ExecBase    =       4
OpenLib     =       -552
CloseLib    =       -414
Write       =       -48
Output      =       -60

        movem.l  a0,-(sp)        ; Kommandozeile sichern

        move.l   ExecBase,a6     ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

        jsr      Output(a6)      ; Output-Handle holen
        move.l   d0,d4

        movem.l  (sp)+,a0        ; Kommandozeile zurückholen
        clr.l    d0
        move.b   (a0),d0         ; Zeichen nach d0

* Zeichen in Tabelle suchen

        lea      table1,a0       ; Start der Zeichen-Tabelle
        lea      table1end,a1    ; Ende der Zeichen-Tabelle
        bclr     #5,d0           ; Force Uppercase
        clr.l    d1              ; Zähler löschen
lab1:   cmp.b    (a0)+,d0        ; Zeichen gefunden?
```

```
        beq     lab2            ; Wenn ja
        addq    #1,d1           ; Sonst Zähler erhöhen
        cmp.l   a1,a0           ; Tabellenende erreicht?
        bne     lab1            ; Wenn nein

* Zeichen nicht gefunden

        lea     text5,a0        ; 'Nicht gefunden'-Text
        bsr     print           ; Ausgeben
        bra     ende

* Einsprungadresse holen

lab2:   lea     table2,a0       ; Tabellenanfang
        asl     #2,d1           ; d1 mal 4
        move.l  0(a0,d1),a1     ; Einsprungadresse nach a1
        jsr     (a1)            ; Routine anspringen

ende:   move.l  a6,a1           ; Lib schließen
        move.l  4,a6
        jsr     CloseLib(a6)

        rts                     ; Zum CLI

load:   lea     text1,a0        ; Start von Text 1 laden
        bsr     print           ; Zur Schreib-Subroutine
        rts

save:   lea     text2,a0
        bsr     print
        rts

print:  lea     text3,a0
        bsr     print
        rts

quit:   lea     text4,a0
        bsr     print
        rts

print:  move.l  d4,d1           ; Output-Handle
        clr.l   d3              ; Länge löschen
        move.b  (a0),d3         ; Erstes Textzeichen = Länge
        lea     1(a0),a1        ; Neuer Textbeginn
        move.l  a1,d2           ; nach d2
        jsr     Write(a6)
        rts

* Datenbereich

dosname:        dc.b    "dos.library",0
                even

table1:         dc.b    "L","S","D","E"
table1end:
```

```
table2:         dc.l    load,save,print,quit

text1:          dc.b    19,"Menüauswahl: Laden",10
                even
text2:          dc.b    23,"Menüauswahl: Speichern",10
                even
text3:          dc.b    21,"Menüauswahl: Drucken",10
                even
text4:          dc.b    18,"Menüauswahl: Ende",10
                even
text5:          dc.b    24,"Zeichen nicht gefunden!",10
                even
```

Programm 3.6: Mehrfachverzweigung mit zwei Tabellen

Also nochmal: Start und Ende der Zeichentabelle werden mit

```
        lea     table1,a0       ; Start der Zeichen-Tabelle
        lea     table1end,a1    ; Ende der Zeichen-Tabelle
```

in Adreßregister geholt. Das Zeichen wird in einen Großbuchstaben umgewandelt. Übrigens: Falls Sie sowohl Groß- als auch Kleinbuchstaben in Ihrem Menü haben, können Sie das Force Uppercase natürlich auch weglassen. Es hat hier nur den Zweck, daß große und kleine Buchstaben gleich behandelt werden.

Dann wird der Tabellenzähler d1 gelöscht. In einer Schleife wird die Tabelle nach dem gewünschten Zeichen durchsucht:

```
lab1:   cmp.b   (a0)+,d0        ; Zeichen gefunden?
```

Wenn es gefunden wird, geht es nach 'lab2', ansonsten wird der Zähler erhöht und zum Schleifenanfang zurückgesprungen, wenn das Tabellenende noch nicht erreicht wurde:

```
        cmp.l   a1,a0           ; Tabellenende erreicht?
        bne     lab1            ; Wenn nein
```

Wurde das Zeichen nicht gefunden, wird ein entsprechender Text ausgegeben und anschließend zum Programmende verzweigt:

```
        lea     text5,a0        ; 'Nicht gefunden'-Text
        bsr     print           ; Ausgeben
        bra     ende
```

Bei erfolgreicher Suche steht in d1 die Tabellenplatznummer des Zeichens. Die Nummer wird mit 4 malgenommen (Langwort für Adresse), und über Nummer und Start der Adreßtabelle wird die Einsprungadresse der Routine geholt:

```
lab2:   lea     table2,a0       ; Tabellenanfang
        asl     #2,d1           ; d1 mal 4
```

```
move.l  0(a0,d1),a1     ; Einsprungadresse nach a1
jsr     (a1)            ; Routine anspringen
```

Danach folgt das Programmende (Lib schließen und RTS).
Nun noch eine kurze Erklärung zum Print-Unterprogramm: Es erwartet, wie gesagt, in a0 einen Zeiger auf den Text, wobei die Länge im ersten Byte des Textes steht. Das Output-Handle wird in d4 vorausgesetzt (dorthin hatten wir es ja gerettet) und nach d1 geholt. Das Textlängen-Register d3 wird zuerst long-gelöscht, dann wird die Länge byte-weise hineingeschrieben. Nun folgt ein adreß-erhöhender LEA-Befehl:

```
lea     1(a0),a1        ; Neuer Textbeginn
```

Hier hätten wir auch den ADD-Befehl auf a0 verwenden können, aber LEA ist schneller und kürzer. Der neue Textbeginn in a1 kommt ins Write-Textbeginn-Register d2, dann wird Write aufgerufen.

Zum Abschluß dieses Kapitels werden wir das versprochene CLI-Taschenrechner-Programm entwickeln.

3.9 Abschluß-Programm: CLI-Taschenrechner

Das zum Programm gehörende Listing finden Sie nur auf der Diskette (weil es zum Abdrucken etwas zu lang ist). Hier wollen wir seine Funktionsweise beschreiben und auf Neuheiten eingehen.

In diesem Kapitel haben wir uns hauptsächlich mit Zahlenumrechnungen und Verzweigetechniken beschäftigt. Daher soll das Abschluß-Programm ein kleiner Taschenrechner für den CLI sein. Er soll folgende Funktionen haben:

1. Die Kommandozeile ist in bis zu drei Parameter, getrennt durch Leerzeichen zu zerlegen. Der erste Parameter ist die erste Zahl, der zweite die Rechenoperation und der dritte die zweite Zahl.
2. Die Parameter 2 und 3 können auch wegfallen.
3. Die Zahlen können dezimal oder hexadezimal sein (bei hex muß ein '$' davor stehen).
4. Die Rechenoperation kann, falls vorhanden, '+','-','*' oder '/' (für Addition, Subtraktion, Multiplikation oder Division) sein. Falls die Parameter 2 und 3 fehlen, wird nicht gerechnet.
5. Das Ergebnis wird in dezimal und hexadezimal ausgegeben. Die Möglichkeit, keine Operation und zweite Zahl anzugeben, dient dazu, eine Zahl nur umzurechnen (dez-hex oder hex-dez).

Den Sourcecode finden Sie auf der Diskette im Verzeichnis "KAPITEL_3" unter dem Namen "PRG_3_7.S".

Fangen wir mit den Unterprogrammen an. Als erstes kommt eine Routine, die die Kommandozeile in ihre Einzelteile "zerlegt" (ein sog. Parser). Sie hat die Aufgabe, den Teil der Kommandozeile bis zum nächsten Leerzeichen oder zum Zeilenende (Return-Code) in einen Puffer zu schreiben. In a0 erwartet sie den Zeiger auf den Quelltext (also die Kommandozeile), in a1 den Ziel-Zeiger und in d0 die maximale Länge des Zieltextes. d0 dient als Schleifenzähler, wird also zunächst um eins erniedrigt:

```
parser: subq     #1,d0
```

Dann wird immer ein Zeichen aus dem Quell- in den Zieltext kopiert, und zwar solange, bis ein Leerzeichen oder Return übertragen wurde oder d0 abgelaufen ist:

```
ps1:    move.b   (a0),(a1)+
        cmp.b    #" ",(a0)
        beq      ps2
        cmp.b    #10,(a0)
        beq      ps2
        add.l    #1,a0
        dbra     d0,ps1
```

Danach wird das letzte übertragene Zeichen, das entweder ein Leerzeichen oder ein Return war, durch ein Nullbyte ersetzt (im Hinblick auf spätere Benutzung der Umwandlungsroutinen):

```
ps2:    move.b   #0,-1(a1)       ; Letztes Zeichen durch 0-Byte ersetzen
```

Wenn der Quellzähler jetzt auf einem Return steht, welches das Ende der Kommandozeile anzeigt, bleibt er unverändert. Ansonsten wird er um eins erhöht, da er in diesem Fall auf einem Trenn-Leerzeichen stand, welches beim nächsten Parseraufruf nicht als Startzeichen vorkommen darf.

Damit ist die Parser-Routine klar. Die vier Zahlen-Umrechnungsroutinen sind auch schon bekannt, deshalb kommen wir nun zur neuen, etwas komfortableren Print-Subroutine, die wir auch später des öfteren verwenden werden:

```
print:  movem.l  d1-d3,-(sp)     ; Register sichern
        move.l   a0,d2           ; Text-Start nach d2 (für Write)
        clr.l    d3              ; Länge löschen
pr1:    addq     #1,d3           ; Länge plus 1
        tst.b    (a0)+           ; Textende-Kennzeichen erreicht?
        bne      pr1             ; Wenn nein
        subq     #1,d3           ; Letzten addq rückgängig

        move.l   d4,d1           ; Output-Handle wird in d4 erwartet
        jsr      Write(a6)       ; Write aufrufen
        movem.l  (sp)+,d1-d3     ; Register zurück
        rts                      ; Bye bye
```

Bisher mußten wir die Länge unserer Texte immer abzählen. Ab jetzt sparen wir uns das. Wir schreiben einfach ein Null-Byte als Endekennzeichen hinter den Text und lassen die Print-Routine die Zeichen bis zum Null-Byte zählen. Die Routine erwartet in a0 den Textbeginn, welchen wir gleich nach d2 schreiben (wo Write ihn erwartet). Die Länge löschen wir zunächst. In der Schleife

```
pr1:    addq    #1,d3           ; Länge plus 1
        tst.b   (a0)+           ; Textende-Kennzeichen erreicht?
        bne     pr1             ; Wenn nein
```

erhöhen wir d3 und a0 solange, bis wir das Null-Byte finden. Da wir d3 einmal zuviel erhöht haben (die Erhöhung kommt vor der Abfrage, und das Null-Byte gehört nicht mehr zum Text selbst) müssen wir wieder eins abziehen:

```
        subq    #1,d3           ; Letzten addq rückgängig
```

Die Parameter d2 und d3 (Start und Länge) sind nun schon richtig gesetzt, nur das Filehandle in d1 fehlt noch. Wir lassen die Routine das gewünschte Handle in d4 voraussetzen, von wo aus es nach d1 kopiert wird. Dann wird Write aufgerufen und die Sache ist gegessen.

Nun kommen wir zum Hauptprogramm. Bis zur Zeile

```
        move.l  (sp)+,a0        ; Kommandozeile zurück
```

ist noch aller klar. Die folgende Abfrage, ob eine Kommandozeile vorhanden ist, nützt die Tatsache, daß das Return-Zeichen, welches eine CLI-Eingabe abschließt, auch immer als letztes Zeichen in der Kommandozeile steht. Falls nun das erste Zeichen das Return ist, springen wir sofort zum Programmende, da keine Eingabe erfolgt ist:

```
        lea     txt5,a0         ; Text "Erste Zahl fehlt"
        bsr     print           ; ausgeben
        bra     ende            ; Zum Programmende
```

Als nächstes wird die Parser-Routine angesprungen und die erste eingegebene Zahl im Puffer 'zahl1' untergebracht. Falls dann das Return-Zeichen gefunden sein sollte, gibt es nur die eine Zahl, kein Rechenzeichen und keine zweite, weshalb wir das Parsen dieser beiden Teile überspringen. Ansonsten werden Rechenzeichen und zweite Zahl in ihre Puffer 'zeichen' und 'zahl2' geparst.

Danach müssen die Zahlen, die als ASCII-Text vorliegen, in Binärzahlen umgerechnet werden. Dazu wird mit

```
        cmp.b   #"$",(a0)       ; Beginnt Zahl mit "$"?
```

geprüft, ob das erste Zeichen der Zahl ein '$' ist. In diesem Fall wird die ASCII-Hex-Umrechnungsroutine aufgerufen (wobei das erste Zeichen natürlich übersprungen wird), ansonsten die ASCII-Dez-Routine. Das wird für beide Zahlen durchgeführt, bzw. nur für die erste, wenn es keine zweite gibt. Die Zahlen werden in d6 und d7 abgelegt.

Dann wird getestet, ob ein Rechenzeichen vorhanden ist, ob also überhaupt gerechnet werden muß. Wenn dies der Fall ist, wird das eingegebene Rechenzeichen in der ersten Tabelle gesucht:

```
        lea     tab1,a0         ; Suche Rechenzeichen in der
        lea     table,a1        ; Zeichen-Tabelle
        move.b  zeichen,d0
        clr.l   d1
ts1:    cmp.b   (a0)+,d0
        beq     ts2
        addq    #1,d1
        cmp.l   a1,a0
        bne     ts1
        lea     txt1,a0         ; Text "Illegales Rechenzeichen"
        bsr     print           ; Ausgeben
        bra     ende            ; Zum Programmende
ts2:    asl     #2,d1           ; Einsprung der Rechen-Routine holen
```

Wird es nicht gefunden, wurde ein ungültiges Zeichen eingegeben, was mit einem entsprechenden Text quittiert wird. Ansonsten wird aus der Tabellenplatznummer des Rechenzeichens die Platznummer der Tabelle der Einsprungadressen bestimmt (Mehrfachverzweigung mit zwei Tabellen) und die Routine aufgerufen:

```
        lea     tab2,a0
        move.l  0(a0,d1),a1
        jsr     (a1)            ; Anspringen
```

Es gibt vier Routinen, welche die jeweils gewünschte Rechenoperation auf die Register d6 und d7 ausführen. Danach wird die Ergebniszahl sowohl in Hex als auch in Dezimal ausgegeben (Aufruf der Umrechnungsroutinen) und das Programm ist beendet.

Kapitel 4
Die DOS-Library

Grundlegende Dateifunktionen

Kommentare und Schutzstatus

Verzeichnis-Operationen

Datenstrukturen in Assembler

Arbeit mit DOS-Fenstern

CON- und RAW-Fenster

Sonstige DOS-Geräte

Ausführung von CLI-Befehlen und Stapeldateien

Laden von Programmen als Segmente

Multitasking-gerechtes Warten

Die Libraries sind, wie eingangs schon erwähnt, die Grundlage und wichtigste Einrichtung im Amiga-System. Daher wollen wir nun jeweils ein Kapitel einer bestimmten Library widmen. Wir beginnen mit der DOS-Lib (die am einfachsten zu handhaben ist) und kommen später zu Intuition, Diskfont, Graphics und Exec. Außerdem wird noch ein Kapitel "DOS für Fortgeschrittene" folgen, in dem all das erklärt wird, was eigentlich auch zur DOS-Library gehört, aber zu diesem Zeitpunkt noch zu kompliziert ist (Prozeß-Strukturen, Filehandle-Strukturen usw.).

Bisher haben wir die DOS-Library nur dazu benutzt, Texte im CLI-Fenster auszugeben und Zeichen von der Tastatur zu lesen. Das ist natürlich noch lange nicht alles, was sie zu bieten hat, aber mit Read und Write haben wir schon zwei sehr wichtige Funktionen benutzt. Die DOS-Lib ist, wie die Bezeichnung "Disk Operating System" schon sagt, hauptsächlich für die Verwaltung von Speichermedien (Disketten, Festplatten usw.) bzw. den darauf enthaltenen Dateien und Verzeichnissen zuständig. Für das Amiga-DOS ist der Begriff "Datei" aber nicht streng an eine Diskettendatei gebunden. Auch einfache Fenster, der Drucker, die "Serielle Schnittstelle" und sogar das Sprachausgabe-System können als Datei angesprochen werden.

4.1 Grundlegende Datei-Operationen

Zu Beginn wollen wir uns mit grundlegenden Datei-Operationen wie Öffnen, Schließen, Lesen, Schreiben usw. beschäftigen und uns dabei nur auf Disk-Dateien beziehen. Die Ansteuerung von Fenste   von Fenstern, Drucker usw. kommt später.

4.1.1 Öffnen und Schließen

Bevor man mit einer Datei arbeiten kann, muß man deren Benutzung quasi beim System anmelden. Man nennt das "Öffnen" der Datei. Ebenso muß man die Datei wieder abmelden, sie also "schließen", wenn sie nicht mehr benötigt wird. Dazu gibt es zwei Routinen in der DOS-Library.

Zum Öffnen einer Datei verwendet man die DOS-Routine Open:

Open	=	**-30 (DOS-library)**

***name**	**d1**	<	Zeiger auf Namenstext der Datei (0-terminiert)
mode	**d2**	<	Öffnungs-Modus
***file**	**d0**	>	Handle der Datei oder 0 bei Fehler
Erklärung			Öffnet eine Datei

Das '0-terminiert' bedeutet, daß der Namenstext (mit 'dc.b' im Programm abgelegt) mit einem Nullbyte enden muß. Generell müssen alle Namen und ähnliches, die per Pointer an Library-Routinen übergeben werden, mit einem Nullbyte enden (es sei denn, es wird auch die Länge des Namens übergeben).

Öffnungs-Modus: Man kann in d2 verschiedene Zahlen eintragen, welche die Art der zu öffnenden Datei bestimmen. Diesen Zahlen werden in den Assembler-Include-Files Texte zugewiesen, aus denen die Bedeutungen der Zahlen besser ersichtlich sind. Die Texte werden von Commodore vorgegeben und sind "standardisiert". Immer, wenn bestimmte Zahlen zur Auswahl irgendwo eingetragen werden müssen, stellen wir sie in zwei Tabellen vor. In der ersten Tabelle stehen die symbolischen Texte mit ihren zugehörigen konkreten Zahlen und einer kurzen Erklärung ihrer Bedeutungen und in der zweiten die ausführlichen Beschreibungen. Hier nun die Tabellen der Öffnungs-Modi:

Öffnungs-Modus	Wert	Bedeutung
MODE_OLDFILE	1005	Öffnet bestehende Datei
MODE_READWRITE	1004	Öffnet neue oder bestehende Datei
MODE_NEWFILE	1006	Erstellt neue Datei

MODE_OLDFILE	Öffnet eine schon bestehende Datei zum Lesen und Schreiben. Sollte es die Datei nicht geben, wird ein Fehler gemeldet.
MODE_READWRITE	Öffnet eine Datei zum Lesen und Schreiben. Wenn die Datei schon existiert, wird die vorhandene Version benutzt, falls nicht, wird sie neu angelegt.
MODE_NEWFILE	Legt eine neue Datei an und öffnet sie zum Lesen und Schreiben. Falls es die Datei schon gab, wird sie gelöscht und durch die Neue ersetzt. Das aufrufende Programm erhält die alleinige Zugriffsberechtigung. Andere Programme können die Datei so lange nicht öffnen, bis sie vom aufrufenden Programm geschlossen wurde.

Über diese Öffnungs-Modi herrscht offensichtlich ziemliche Verwirrung. Wir haben in jeder Dokumentation eine leicht unterschiedliche Erklärung gefunden. Die von uns deklarierten Erläuterungen haben wir allerdings ausprobiert, sie sollten also wirklich stimmen.

In d0 steht nach dem Aufruf das File-Handle der geöffneten Datei oder eine 0, wenn ein Fehler auftritt. Das Handle, das wir bei der Textausgabe im CLI-Fenster schon benutzt haben, ist auch hier eine "Identifikationsnummer" der Datei. Bei vielen weiteren Operationen, die wir mit der Datei durchführen, müssen wir das Handle angeben. (Eigentlich handelt es sich dabei um einen Zeiger auf die FileHandle-Struktur, die im DOS-Fortgeschrittenen-Kapitel besprochen wird.) Falls die Datei nicht geöffnet werden konnte, kann

man sich mit einer weiteren Funktion genauere Informationen über aufgetretene Fehler besorgen.

Für jede geöffnete Datei gibt es einen sog. "Datenzeiger", der auf die Position des nächsten zu lesenden bzw. zu schreibenden Bytes zeigt. Unmittelbar nach dem Öffnen steht er immer am Dateianfang. Bei jedem Zugriff auf die Datei wird der Datenzeiger um so viele Bytes zum Dateiende hin verschoben, wie gelesen bzw. geschrieben wurden. Es sei denn, es soll über das Dateiende hinaus gelesen werden, dann werden nur die vorhandenen Bytes gelesen, ohne daß ein Fehler gemeldet wird. Wichtig ist auch, daß die Datei immer gelesen und geschrieben werden kann, egal, wie sie geöffnet wurde. Wenn der Datenzeiger nicht am Dateiende steht, also auf schon vorhandene Daten zeigt, werden diese bei einem Schreibzugriff durch die neuen Daten ersetzt.

Wenn man die Datei nicht mehr benötigt, muß sie geschlossen werden. Dieses Schließen ist vor allem dann sehr wichtig, wenn man Daten in die Datei schreibt, da diese zunächst nur in einen bestimmten Speicherbereich geschrieben werden (in den sog. "Dateipuffer"). Der Puffer wird erst dann wirklich auf die Diskette gebracht, wenn er voll ist oder die Datei geschlossen wurde. Wenn Sie also Daten in eine Datei schreiben, die den Puffer nicht ganz füllen, die Datei später aber nicht schließen, bekommt die Diskette die Daten niemals zu sehen.

Das Schließen einer Datei übernimmt die DOS-Routine Close:

Close	=	**-36 (DOS-library)**

***file**	**d1**	<	Handle der zu schließenden Datei
Erklärung			Schließt eine Datei

Die Close-Routine bekommt nur einen Parameter (das Handle der Datei, das wir bei Open bekamen) und liefert keinen Rückgabewert.

4.1.2 Abfrage von Fehlern

Zur Behandlung von Fehlern gibt es die DOS-Routine IoErr. Immer, wenn eine DOS-Routine einen Rückgabewert liefert, der das Auftreten eines Fehlers anzeigt, kann man mit IoErr genauere Informationen darüber abholen, was eigentlich schief gelaufen ist:

IoErr	=	-132 (DOS-Library)

error	**d0**	>	DOS-Fehlercode
Erklärung			Ermittelt den DOS-Fehlercode

Die wichtigsten Fehlercodes, die bei Einsteigern auftreten können, haben folgende Bedeutungen:

```
ERROR_NO_FREE_STORE             equ     103
ERROR_FILE_NOT_OBJECT           equ     121
ERROR_OBJECT_IN_USE             equ     202
ERROR_OBJECT_EXISTS             equ     203
ERROR_OBJECT_NOT_FOUND          equ     205
ERROR_OBJECT_WRONG_TYPE         equ     212
ERROR_DISK_NOT_VALIDATED        equ     213
ERROR_DISK_WRITE_PROTECTED      equ     214
ERROR_RENAME_ACROSS_DEVICES     equ     215
ERROR_DIRECTORY_NOT_EMPTY       equ     216
ERROR_DEVICE_NOT_MOUNTED        equ     218
ERROR_SEEK_ERROR                equ     219
ERROR_COMMENT_TOO_BIG           equ     220
ERROR_DISK_FULL                 equ     221
ERROR_DELETE_PROTECTED          equ     222
ERROR_WRITE_PROTECTED           equ     223
ERROR_READ_PROTECTED            equ     224
ERROR_NOT_A_DOS_DISK            equ     225
ERROR_NO_DISK                   equ     226
ERROR_NO_MORE_ENTRIES           equ     232
```

ERROR_NO_FREE_STORE	Nicht genug Speicherplatz.
ERROR_FILE_NOT_OBJECT	Sie haben versucht, eine Textdatei o.ä. als Programm zu starten.
ERROR_OBJECT_IN_USE	Objekt (Datei, Verzeichnis usw.) wird schon von einem anderen Programm benutzt, das sich die Exklusivrechte geben ließ.
ERROR_OBJECT_EXISTS	Das Objekt existiert schon.
ERROR_OBJECT_NOT_FOUND	Das Objekt wurde nicht gefunden.
ERROR_OBJECT_WRONG_TYPE	Der Objekts-Typ stimmt nicht. Sie haben z.B. versucht, ein Verzeichnis als Datei zu öffnen.
ERROR_DISK_NOT_VALIDATED	Die angesprochene Diskette wurde noch nicht vom DOS als gültig erklärt.
ERROR_DISK_WRITE_PROTECTED	Die Diskette ist schreibgeschützt.
ERROR_RENAME_ACROSS_DEVICES	Mittels RENAME kann man eine Datei lediglich in ein anderes Verzeichnis, nicht aber auf eine andere Diskette bringen.
ERROR_DIRECTORY_NOT_EMPTY	Nur ein leeres Verzeichnis kann gelöscht werden.
ERROR_DEVICE_NOT_MOUNTED	Das angesprochene Gerät ist dem DOS nicht bekannt.
ERROR_SEEK_ERROR	Der SEEK-Befehl wurde über die Dateigrenzen hinaus angewandt.
ERROR_COMMENT_TOO_BIG	Der angegebene Dateikommentar ist zu lang.

ERROR_DISK_FULL	Die Diskette ist voll.
ERROR_DELETE_PROTECTED	Eine löschgeschützte Datei sollte gelöscht werden.
ERROR_WRITE_PROTECTED	Eine schreibgeschützte Datei sollte verändert werden.
ERROR_READ_PROTECTED	Eine lesegeschützte Datei sollte gelesen werden.
ERROR_NOT_A_DOS_DISK	Die angesprochene Diskette ist keine gültige DOS-Diskette.
ERROR_NO_DISK	Im angegebenen Laufwerk ist keine Diskette.
ERROR_NO_MORE_ENTRIES	Das untersuchte Verzeichnis enthält keine weiteren Einträge.

Falls Sie den einen oder anderen Fehler nicht verstehen sollten, macht nichts, später gehen wir noch genauer darafu ein. Die Tabelle steht nur der Vollständigkeit halber in dieser Form hier. Anhand des Fehlercodes, den Sie bekommen, können Sie dann im Programm entsprechend reagieren. Im Falle "Write Protected" könnten Sie z.B. den Benutzer auffordern, den Schreibschutz zu entfernen, bei "Disk Full" oder "No Disk" möchte er doch bitte eine passende Diskette einlegen, usw.

Als nächstes ein kleines Beispielprogramm zum Öffnen und Schließen einer Datei und zur Fehlerbehandlung. Die Datei 'test' soll in der RAM-Disk erzeugt und geöffnet werden. Wenn ein Fehler auftritt, wird in eine entsprechende Routine verzweigt, ansonsten wird die Datei wieder geschlossen. Die DOS-Basis wird in a6 vorausgesetzt:

```
        ...
        move.l  #fname,d1       ; Zeiger auf Namenstext nach d1
        move.l  #1006,d2        ; Öffnungs-Modus: Neue Datei
        jsr     -30(a6)         ; Open anspringen
        tst.l   d0              ; Fehler beim Öffnen?
        bne     lab1            ; Ungleich 0 = kein Fehler

        jsr     -132(a6)        ; DOS-Routine IoErr
        move.l  d0,d2           ; Fehlercode zur Weiterverarbeitung
                                ; nach d2
        bra     error           ; Zur Fehlerbehandlungs-Routine

lab1:   move.l  d0,d4           ; Handle in d4 sichern
        ...                     ; Weitere Programmteile

        move.l  d4,d1           ; File-Handle nach d1
        jsr     -36(a6)         ; Zur Close-Routine

        ...                     ; Und weiter im Programm

fname:        dc.b    "ram:test",0
              even
```

Bild 4.1: Beispielprogramm Öffnen und Schließen einer Datei

4.1.3 Lesen und Schreiben

Nun können wir also Dateien öffnen und schließen. Das alleine reicht noch nicht. Wir wollen ja auch irgendetwas mit der Datei anstellen, sprich Daten in ihr speichern. Die DOS-Routinen, die wir dazu benötigen (Read und Write), wurden im 3. Kapitel schon ausführlich vorgestellt, deshalb hier nur noch einmal der "Library-Routinen-Kasten":

Read	**=**	**-42 (DOS-Library)**

***file**	**d1**	<	Handle der zu lesenden Datei
***buffer**	**d2**	<	Startadresse des Lese-Puffers
length	**d3**	<	Anzahl Datenbytes
reallyread	**d0**	>	Anzahl wirklich gelesener Bytes (-1=Fehler)
Erklärung			Liest Daten aus einer Datei

Write	**=**	**-48 (DOS-Library)**

***file**	**d1**	<	Handle der zu schreibenden Datei
***buffer**	**d2**	<	Startadresse der Daten
length	**d3**	<	Anzahl Datenbytes
writtenOut	**d0**	>	Anzahl wirklich geschriebener Bytes (-1=Fehler)
Erklärung			Schreibt Daten in eine Datei

Da die beiden Routinen schon bekannt sind, können wir gleich das Beispiel-Programm erläutern. Wir wollen die Kommandozeile in eine RAM-Testdatei schreiben, sofort wieder aus ihr lesen und auf den Bildschirm schreiben. Hinweis: Zur Benutzung der RAM-Disk muß sich die Datei "ram-handler" im L-Verzeichnis Ihrer Startdiskette befinden!

```
* Programm 4.1: Kommandozeile in Datei schreiben

ExecBase    =    4
OpenLib     =    -552
CloseLib    =    -414
Output      =    -60
Open        =    -30
Close       =    -36
Read        =    -42
Write       =    -48
```

```
        movem.l  a0/d0,-(sp)      ; Kommandozeile sichern

        move.l   ExecBase,a6      ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

        jsr      Output(a6)       ; Output-Handle holen
        move.l   d0,d4

        move.l   #fname,d1        ; RAM-Datei öffnen
        move.l   #1006,d2         ; Neue Datei
        jsr      Open(a6)
        tst.l    d0
        bne      lab1

* Fehler aufgetreten

        add.l    #8,sp            ; Kommandozeile vom Stack löschen
        lea      fehtext,a0       ; Text 'Fehler aufgetreten'
        bsr      print            ; Im Window ausgeben
        bra      ende

* Kein Fehler: Daten in Datei schreiben

lab1:   move.l   d0,d5            ; Handle der RAM-Datei in d5 sichern
        movem.l  (sp)+,a0/d0      ; Kommandozeile zurückholen
        move.l   d5,d1            ; Handle der Schreib-Datei nach d1
        move.l   a0,d2            ; Startadresse der Daten (Kommandoz.)
        move.l   d0,d3            ; Länge der Daten
        jsr      Write(a6)        ; Write aufrufen

* RAM-Datei wieder schließen

        move.l   d5,d1            ; Handle der RAM-Datei
        jsr      Close(a6)        ; Datei schließen

* Jetzt existierende RAM-Datei öffnen

        move.l   #fname,d1        ; Datei-Name
        move.l   #1005,d2         ; Existierende Datei
        jsr      Open(a6)         ; Datei öffnen
        move.l   d0,d5            ; Handle sichern

* Text aus RAM-Datei lesen

        move.l   d5,d1            ; Handle nach d1
        move.l   #buff,d2         ; Adresse des Puffers
        move.l   #40,d3           ; Maximal 40 Zeichen
        jsr      Read(a6)         ; Lese-Aufruf
        move.l   d0,d3            ; Rückgabe: Anzahl wirklich gelesener
                                  ; Bytes in d3 sichern
```

```
* Gelesene Daten in Window schreiben

        lea     buff,a0         ; Start des Puffers
        move.b  #0,0(a0,d3)     ; Text-Endekennzeichen: 0-Byte
        bsr     print           ; Ausgabe-Subroutine

        move.l  d5,d1           ; RAM-Datei schließen
        jsr     Close(a6)

ende:   move.l  a6,a1           ; Lib schließen und Ende
        move.l  ExecBase,a6
        jsr     CloseLib(a6)

        rts

print:  movem.l d1-d3,-(sp)     ; SUB Textausgabe für DOS-Write
        move.l  a0,d2           ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l   d3              ;  d4 < Handle der Ausgabedatei
pr1:    addq    #1,d3
        tst.b   (a0)+
        bne     pr1
        subq    #1,d3

        move.l  d4,d1
        jsr     Write(a6)
        movem.l (sp)+,d1-d3
        rts

* Datenbereich

dosname:        dc.b    "dos.library",0
                even
fname:          dc.b    "ram:test",0
                even
buff:           ds.b    40
fehtext:        dc.b    42,"Fehler beim Öffnen der Datei aufgetreten!",10
```

Programm 4.1: Kommandozeile in Datei schreiben

Im Programmteil 'Fehler aufgetreten' wird die Kommandozeile auf eine neue, etwas ungewohnte Weise vom Stack gelöscht. Da sie nicht mehr gebraucht wird, ist es nicht nötig, sie beim Herunternehmen vom Stack noch in ein Register zu schreiben. Alle Daten, die man auf den Stack gelegt hat, muß man allerdings vor dem Programmende auch wieder von dort entfernen, weil sie sonst fälschlicherweise als Return-Adresse interpretiert würden. Eine solche Adresse würde aber sicherlich nicht zum CLI zurückführen, sondern höchstens ins Reich des Gurus. Da wir die Daten nicht in einem Register ablegen müssen, erhöhen wir nur den Stackpointer durch einen ADD-Befehl. Wir addieren also 8 zum Stackpointer, wodurch 8 Bytes oder 2 Langworte vom Stack verschwunden sind.

Die Benutzung von Read und Write dürfte eigentlich klar sein. Ein paar Worte noch zur folgenden Stelle:

```
        move.l   #40,d3         ; Maximal 40 Zeichen
jsr     Read(a6)        ; Lese-Aufruf
move.l  d0,d3           ; Rückgabe: Anzahl wirklich gelesener
                        ; Bytes in d3 sichern
```

Unsere eigene Print-Routine erwartet ja als Endekennzeichen des Textes ein Nullbyte. Die Datenlänge in d3 ist hier als Maximalwert anzusehen. Nach dem Read steht in d0 die Anzahl wirklich gelesener Bytes (falls die Datei kleiner war als in d3 angegeben). Diese Anzahl wird in d3 gesichert. Ein paar Befehle später wird mittels

```
lea     buff,a0         ; Start des Puffers
move.b  #0,0(a0,d3)     ; Text-Endekennzeichen: 0-Byte
```

das Nullbyte hinter den Text geschrieben. Wir verwenden die Adressierungsart ARI mit Index und Offset, wobei wir den Offset nicht benötigen. Zur Startadresse des Textpuffers in a0 wird die Textlänge in d3 hinzuaddiert, wodurch wir genau ein Byte hinter dem letzten Textzeichen landen. Dort wird das Nullbyte amgefügt.

Öfters vorkommende Subroutinen, wie z.B. die Print-Routine werden wir nur beim ersten Auftreten komplett kommentieren, in den weiteren Programmen werden wir dann nur noch ihren Zweck und die zu übergebenden Parameter angeben.

4.1.4 Versetzen der Lese/Schreib-Position

Die Routinen Read und Write bearbeiten eine Datei "sequentiell", d.h. man kann sie immer nur "am Stück" vom Anfang bis zum Ende lesen oder beschreiben. Wenn Sie z.B. das 50. Byte einer Datei lesen wollen, müssen Sie, wenn Sie mit Read arbeiten, zuerst die Bytes 1-49 lesen, um an das 50. heranzukommen. Dieser Umstand kann, vor allem bei längeren Dateien, zu unnötigem Leseaufwand führen. Es gibt daher eine DOS-Routine, die Abhilfe schafft. Sie heißt Seek und dient zum Versetzen des Datenzeigers, der ja immer auf das nächste zu lesende oder zu schreibende Byte einer Datei zeigt:

Seek	= -66 (DOS-Library)

***file**	**d1**	<	Handle der Datei, deren Datenzeiger versetzt werden soll
pos	**d2**	<	Anzahl der Bytes, um die der Datenzeiger versetzt wird
offset	**d3**	<	Gibt an, wie 'pos' zu interpretieren ist
oldpos	**d0**	>	Vorige Position des Datenzeigers relativ zum Dateianfang

Erklärung Versetzt den Datenzeiger einer Datei

Folgende Werte für 'offset' sind möglich:

Offset-Typ	Wert	Bedeutung
OFFSET_BEGINNING	-1	Relativ zum Dateianfang
OFFSET_CURRENT	0	Relativ zur derzeitigen Position
OFFSET_END	1	Relativ zum Ende

OFFSET_BEGINNING Der Datenzeiger wird an den Dateianfang plus Positionszahl, die positiv sein muß, gesetzt.

OFFSET_CURRENT Der Datenzeiger wird relativ zur derzeitigen Position versetzt, und zwar in Richtung Dateiende, wenn 'pos' positiv ist und in Richtung Dateianfang, wenn 'pos' negativ ist.

OFFSET_END Der Datenzeiger wird an das Dateiende minus Positionszahl, die positiv sein muß, gesetzt.

Da uns Seek die alte Position des Datenzeigers zurückgibt, kann man auf einfache Weise die Länge einer geöffneten Datei feststellen, und zwar so:

```
        move.l  d5,d1           ; Das Handle stehe in d5
        move.l  #0,d2           ; Seek nach 0 Bytes
        move.l  #1,d3           ; relativ zum Dateiende
        jsr     -66(a6)

        move.l  d5,d1
        move.l  #0,d2           ; Diesmal nach 0 Bytes
        move.l  #-1,d3          ; relativ zum Anfang
        jsr     -66(a6)

        move.l  d0,d3           ; Länge steht dann in d0
```

Bild 4.2: Dateilänge feststellen mit Seek

Mit dem ersten Seek-Befehl springen wir zum Dateiende. Der zweite führt zum Dateianfang. Wichtig ist nun der Rückgabewert des zweiten Seek: Er entspricht der Datenzeiger-Position vor dem Seek relativ zum Dateianfang. Da diese das Dateiende war, ist sie gleichzeitig die Dateilänge. Einfach, aber wirkungsvoll.

Nun noch ein weiteres Beispiel für Seek-Anwendungen: Aus einer Datei, deren Handle in d5 stehe, sollen ab dem 60. Byte 10 Bytes gelesen werden und dann ab dem 30. Byte 20 Bytes. Hier das Programm:

```
        move.l  d5,d1           ; Handle nach d1
        move.l  #0,d2           ; Seek nach 60 Bytes
        move.l  #-1,d3          ; relativ zum Dateianfang
        jsr     -66(a6)

        bsr     lesen1          ; Fiktive Lese-Routine

        move.l  d5,d1
        move.l  #-40,d2         ; Jetzt -40 Bytes relativ zur
        move.l  #0,d3           ; derzeitigen Position
        jsr     -66(a6)

        bsr     lesen2
        ...
```

Bild 4.3: Weitere Seek-Anwendungen

Der erste Seek wird wohl einleuchten, aber der zweite? Nach dem ersten (fiktiven) Lesen steht der Datenzeiger auf Position 70, da Lese- und Schreibbefehle den Zeiger um so viele Bytes zum Dateiende hin verschieben, wie gelesen (oder geschrieben) wurden. Um dann zum 30. Byte zu kommen, muß der Zeiger um -40 Bytes relativ zur derzeitigen Position verschoben werden. Aller klar?

Übrigens ist ein Seek relativ zur derzeitigen Position immer schneller als ein Seek relativ zum Dateianfang oder -ende.

4.1.5 Umbenennen und löschen

Dies sind zwei ganz einfache Funktionen, und sie sind auch schnell erklärt. Für beide gibt DOS-Routinen, und zwar folgende:

Rename		=	-78 (DOS-Library)
***oldname**	**d1**	<	Zeiger auf Namenstext der umzubenennenden Datei
***newname**	**d2**	<	Zeiger auf neuen Namenstext
success	**d0**	>	0 = Fehler aufgetreten
Erklärung			Gibt einer Datei oder einem Verzeichnis einen neuen Namen

In d1 erwartet die Routine einen Zeiger auf den Namen der Datei, die umbenannt werden soll, und in d2 einen Zeiger auf den neuen Namen. Zu beachten ist, daß mit Rename eine Datei sehr wohl in ein anderes Verzeichnis verlagert werden kann, indem man im neuen Namen einen anderen Pfad angibt. Sollte man allerdings versuchen, eine Datei auf ein anderes Spei-

chermedium zu verlagern, wird Rename mit einer Fehlermeldung (Rename across Devices) abbrechen. Ein kleines Beispiel: Wir haben die Datei 'Test' vom letzten Programm immer noch im RAM stehen. Angenommen, wir haben im RAM ein Unterverzeichnis namens 'Texte', dann bringen wir unser 'Test' auf folgende Weise unter einem neuen Namen dort hinein:

```
        move.l   #oldname,d1     ; Zeiger auf alten Namen
        move.l   #newname,d2     ; Neuer Name
        jsr      -78(a6)         ; Aufruf Rename-Routine
        ...

oldname:      dc.b    "ram:test",0
              even
newname:      dc.b    "ram:texte/test1",0
              even
```

Bild 4.4: Umbenennen einer Datei

Das war's schon. Wenn wir unsere Test-Datei endgültig loswerden wollen, benutzen wir DeleteFile:

DeleteFile	=	**-72 (DOS-Library)**

***name**	**d1**	<	Zeiger auf Namenstext
success	**d0**	>	0 = Fehler aufgetreten
Erklärung			Löscht eine Datei oder ein leeres Verzeichnis

Beispiel: Löschen der Datei "ram:texte/test1":

```
        move.l   #fname,d1       ; Name der Datei
        jsr      -72(a6)         ; Datei löschen
        ...

fname:        dc.b    "ram:texte/test1",0
              even
```

Bild 4.5: Löschen einer Datei

Einfacher geht's wirklich nicht mehr, oder?

4.2 Ordnung ins Chaos: Die Unterverzeichnisse

Unterverzeichnisse sind eine wichtige Einrichtung im Amiga-Dateisystem, denn sie tragen erheblich zur Ordnung auf Ihren Datenträgern bei. Stellen Sie sich nur einmal vor, Sie haben eine Festplatte von 40 MB und schreiben alle Dateien aller darauf befindlichen Programme einfach ins Hauptverzeichnis. Das würde sehr schnell in ein unübersehbares Chaos ausarten. Wir wollen daher als nächstes lernen, wie man in Assembler mit Unterverzeichnissen umgeht, wie man sie anlegt und löscht und wie man das aktuelle Verzeichnis wechseln kann.

4.2.1 Unterverzeichnisse anlegen

Zu diesem Zweck gibt es die DOS-Routine CreateDir, die ähnlich wie die DeleteFile-Routine aufgerufen wird:

CreateDir	=	**-120 (DOS-Library)**

***name**	**d1**	<	Zeiger auf Namen des anzulegenden Verzeichnisses
***newlock**	**d0**	>	Lock des neuen Verzeichnisses oder 0 bei Fehler
Erklärung			Erzeugt ein neues Verzeichnis

Als Name muß der komplette Pfadname des neuen Verzeichnisses angegeben werden, also mit Gerätebezeichnung und mit eventuellen Namen von übergeordneten Verzeichnissen (es sei denn, das Verzeichnis, in dem ein neues angelegt werden soll, ist als das aktuelle Verzeichnis gesetzt). Beispiel: Im Verzeichnis "df0:texte" soll ein Unterverzeichnis namens "sicherung" angelegt werden:

```
        move.l  #dirname,d1
        jsr     -120(a6)
        tst.l   d0              ; Fehler beim Anlegen?
        beq     fehler          ; Wenn ja
        move.l  d0,d5           ; Lock zur Weiterverarbeitung sichern
        ...

dirname:        dc.b    "df0:texte/sicherung",0
                even
```

Bild 4.6: Anlegen eines neuen Verzeichnisses

Sie haben sich sicher schon über den neuen Begriff "Lock" gewundert. Ein "Lock" ist etwas ähnliches wie ein File-

Handle. Es ist eine Art Zugriffsnummer einer Datei oder eines Verzeichnisses, bezieht sich allerdings nicht auf den Inhalt der Datei, sondern dient zum Einholen von Informationen wie Länge, Schutzstatus usw. Hauptsächlich werden Locks beim Einlesen von Verzeichniseinträgen benutzt. Dazu kommen wir später noch, zunächst sind Locks für uns einfach Zugriffsnummern auf ein Verzeichnis.

Genauso, wie Sie eine Datei nach Ende ihrer Bearbeitung schließen müssen, muß auch ein Lock wieder freigegeben werden. Die entsprechende DOS-Routine heißt UnLock:

UnLock	=	**-90 (DOS-Library)**

***lock** **d1** < Freizugebendes Lock

Erklärung Gibt ein Lock wieder frei

Auch das Lock auf ein neu erzeugtes Verzeichnis müssen Sie freigeben (spätestens bei Programmende), es sei denn, Sie wollen es noch irgendwie bearbeiten.

```
        move.l  d5,d1           ; Lock vom Sicherungsregister nach d1
        jsr     -90(a6)         ; UnLock anspringen
```

4.2.2 Unterverzeichnisse umbenennen und löschen

Die Datei-Rename-Routine läßt sich auch auf Verzeichnisse anwenden. Es gelten die selben Regeln wie bei der Datei-Umbenennung (alten und neuen Namen angeben, Verschiebung in ein anderes Verzeichnis möglich, aber nicht auf ein anderes Speichermedium). Benennen wir unser Test-Verzeichnis von vorhin in "sicherung2" um:

```
        move.l  #olddirname,d1 ; Zeiger auf alten Verz.-Namen
        move.l  #newdirname,d2 ; Neuer Name
        jsr     -78(a6)        ; Aufruf Rename-Routine
        ...

olddirname:  dc.b   "ram:texte/sicherung",0
             even
newdirname:  dc.b   "ram:texte/sicherung2",0
             even
```

Bild 4.7: Umbenennen eines Verzeichnisses

Zum Löschen von Verzeichnissen verwenden Sie die selbe Routine wie zum Löschen von Dateien (DeleteFile). Zwei Voraussetzungen muß das Verzeichnis allerdings erfüllen, um gelöscht werden zu können:

1. Es darf zur Zeit nicht "gelockt" sein (weder vom eigenen Programm noch von einem anderen).
2. Es muß völlig leer sein.

Diese Punkte vorausgesetzt können wir unser Test-Verzeichnis jetzt löschen:

```
        move.l  #dirname,d1
        jsr     -72(a6)

dirname:        dc.b    "df0:texte/sicherung2",0
                even
```

Bild 4.8: Löschen eines (leeren) Verzeichnisses

4.2.3 Setzen des aktuellen Verzeichnisses

Der sicherste Weg, eine Datei oder ein Verzeichnis anzusprechen, ist es, den kompletten Pfadnamen anzugeben. Das kann, vor allem bei vielen, geschachtelten Unterverzeichnissen, eine Menge Schreibarbeit bedeuten (das kennen Sie vielleicht vom CLI her). Daher bietet das Dateisystem die Möglichkeit, ein Verzeichnis zum aktuellen Verzeichnis zu erheben. Alle Dateien werden, wenn Sie nur über ihren Namen und ohne Pfadangabe abgesprochen werden, zuerst dort gesucht. In Assembler verwenden wir dafür die DOS-Routine CurrentDir:

CurrentDir		=	**-126 (DOS-Library)**
***lock**	**d1**	<	Lock des zu setzenden aktuellen Verzeichnisses
***oldlock**	**d0**	>	Lock des bisherigen aktuellen Verzeichnisses
Erklärung			Setzt das aktuelle Verzeichnis

Diese Routine erwartet, wie wir sehen, als Parameter ein Lock auf ein Verzeichnis. Das müssen wir uns aber erstmal besorgen, und zwar mit der Routine Lock:

Lock		=	**-84 (DOS-Library)**
***name**	**d1**	<	Zeiger auf Namen des zu lockenden Objekts
type	**d2**	<	Geforderter Typ des Locks
***lock**	**d0**	>	Lock oder 0 bei Fehler
Erklärung			Holt ein Datei- oder Verzeichnis-Lock

'Name' kann ein Verzeichnis oder eine Datei sein (für unser Beispiel CurrentDir natürlich ein Verzeichnis). Für 'type' sind folgende Werte möglich:

Lock-Typ	Wert	Bedeutung
EXCLUSIVE_LOCK	-1	Exklusiv-Recht des aufrufenden Programms
ACCESS_WRITE	-1	Siehe EXCLUSIVE_LOCK
SHARED_LOCK	-2	Kein Exklusiv-Recht
ACCESS_READ	-2	Siehe SHARED_LOCK

EXCLUSIVE_LOCK Exklusiv-Lock-Recht des aufrufenden Programms. Geht nicht, wenn schon ein anderes Programm das Objekt mit EXCLUSIVE_LOCK gelockt hat.

ACCESS_WRITE Identisch mit EXCLUSIVE_LOCK

SHARED_LOCK Andere Programme dürfen das Objekt auch zur gleichen Zeit locken.

ACCESS_READ Identisch mit SHARED_LOCK

Da wir nur das aktuelle Verzeichnis wechseln und nichts schreiben wollen, geben wir uns mit SHARED_LOCK zufrieden. Als Beispiel soll das Verzeichnis "df0:texte" zum aktuellen werden. Nachdem wir das Lock haben, können wir CurrentDir aufrufen. Diese Routine schickt uns als Rückgabewert das Lock des vorigen aktuellen Verzeichnisses, welches wir Un-Locken müssen.

```
        move.l  #dirname,d1     ; Name des Verzeichnisses
        move.l  #-2,d2          ; Modus: SHARED_LOCK
        jsr     -84(a6)         ; Lock holen
        move.l  d0,d4           ; und sichern

        move.l  d0,d1           ; Nach d1 für CurrentDir
        jsr     -126(a6)        ; Dir setzen

        move.l  d0,d1           ; Altes Dir-Lock nach d1
        jsr     -90(a6)         ; und UnLocken
        ...

dirname:        dc.b    "df0:texte",0
                even
```

Bild 4.9: Wechseln des aktuellen Verzeichnisses

So langsam wird es mal wieder Zeit für ein Komplett-Beispielprogramm. Das Thema "aktuelles Verzeichnis" ist dafür auch gut geeignet, denn wir können nun einen CLI-Befehl "nachprogrammieren": den CD-Befehl. Das zu setzende Verzeichnis wird nun aus der Kommandozeile geholt, der Rest bleibt gleich:

```
* Programm 4.2: Selbstprogrammierter CD-Befehl

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
Lock            =       -84
CurrentDir      =       -126
UnLock          =       -90

        movem.l  a0/d0,-(sp)     ; Kommandozeile retten

        move.l   ExecBase,a6     ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

        movem.l  (sp)+,d0/a0     ; Kommandozeile zurück
        move.b   #0,-1(a0,d0)    ; Schluß-Return durch Nullbyte ersetzen

        move.l   a0,d1           ; Kommandozeilenstart nach d1 für Lock
        move.l   #-2,d2          ; Modus SHARED_LOCK
        jsr      Lock(a6)        ; Lock holen

        move.l   d0,d1           ; Lock nach d1 für CurrentDir
        jsr      CurrentDir(a6) ; Dir setzen

        move.l   d0,d1           ; Altes CurrentDir nach d1 für UnLock
        jsr      UnLock(a6)      ; UnLock aufrufen

        move.l   a6,a1           ; DOS-Lib schließen und Ende
        move.l   ExecBase,a6
        jsr      CloseLib(a6)

        rts

* Datenbereich

dosname:     dc.b    "dos.library",0
             even
```

Programm 4.2: Selbstprogrammierter CD-Befehl

Das Programm verhält sich genauso wie der bekannte CD-Befehl. Auch die Angabe eines '/' zum Wechsel in das nächsthöhere Verzeichnis ist möglich. Die Zeile

```
        move.b   #0,-1(a0,d0)    ; Schluß-Return durch Nullbyte ersetzen
```

hat folgenden Sinn: Die Lock-Routine erwartet den, mit einem Nullbyte abgeschlossenen Verzeichnisnamen. In der Kommandozeile steht jedoch als letztes Zeichen ein Return (ASCII-10). Dieses muß durch ein Nullbyte ersetzt werden. Dies ge-

schieht, indem wir an die Adresse 'Kommandozeilenstart plus Kommandozeilenlänge minus 1' eine 0 schreiben. Etwas ähnliches hatten wir ja schon einmal im ersten Programm dieses Kapitels. Folgendes Beispiel soll die Rechnung verdeutlichen: Angenommen, wir haben den Text 'df0:' in die Kommandozeile geschrieben. Diese liege ab Adresse 1000 im Speicher. Ihre Länge beträgt 5 (vier Zeichen plus Return). Das zu ersetzende Return steht also in der Adresse 1004. In unserer Rechnung ergibt sich: 1000 (Startadresse) plus 5 (Länge) minus 1 = 1004, also die richtige Adresse.

4.2.4 Ermitteln des übergeordneten Verzeichnisses

Wenn Sie das Lock eines Verzeichnisses haben, können Sie sich von der Routine ParentDir das Lock des übergeordneten Verzeichnisses geben lassen (also des Verzeichnisses, in dem das erste Verzeichnis steht). Die Amiga-Bezeichnung für ein übergeordnetes Verzeichnis lautet "Parent Directory" (Eltern-Verzeichnis).

ParentDir	**=**	**-210 (DOS-Library)**

***lock**	**d1**	<	Lock des gewünschten Verzeichnisses
***newlock**	**d0**	>	Lock des übergeordneten Verzeichnisses
Erklärung			Holt das Lock des übergeordneten Verzeichnisses

Die Routine gibt Null zurück, wenn kein übergeordnetes Verzeichnis vorhanden ist. Auch hier gilt: Wenn Sie das auf diese Weise ermittelte Parent-Verzeichnis nicht mehr benötigen, müssen Sie es UnLocken.

4.3 Dateikommentar und Schutzstatus

Nun kommen wir zu zwei Routinen, die für Verzeichnisse und Dateien gleichermaßen einsetzbar sind (wenn Sie im folgenden 'Objekt' lesen, ist 'Datei oder Verzeichnis' gemeint). Sie arbeiten nicht mit Locks oder File-Handles, sondern erwarten einfach den Namen des zu bearbeitenden Objekts als Text.

Den Kommentar eines Objektes setzt man mit der DOS-Routine SetComment:

SetComment	=	**-180 (DOS-Library)**

***name**	**d1**	<	Zeiger auf Namen der gewünschten Datei
***comment**	**d2**	<	Zeiger auf Kommentartext (max. 116 Zeichen)
success	**d0**	>	0 = Fehler aufgetreten
Erklärung			Setzt den Kommentar einer Datei oder eines Verzeichnisses

Beispiel: Wir wollen die Datei "df0:test" mit dem Kommentar "Dies ist eine Test-Datei" versehen:

```
        move.l  #filename,d1
        move.l  #comment,d2
        jsr     -180(a6)
        ...

filename:    dc.b    "df0:test",0
             even
comment:     dc.b    "Dies ist eine Test-Datei",0
             even
```

Bild 4.10: Setzen des Datei-Kommentars

Der Schutzstatus einer Datei gibt an, was mit der Datei gemacht werden kann (Lesen, Löschen usw.) und enthält außerdem diverse Attribute (archiviert, Script-Datei usw.). Man verändert ihn mit der Routine SetProtection:

SetProtection	=	**-186 (DOS-Library)**

***name**	**d1**	<	Zeiger auf Dateinamen
mask	**d2**	<	Neuer Schutzstatus
success	**d0**	>	0 = Fehler aufgetreten
Erklärung			Setzt den Datei-Schutzstatus

Im Masken-Langwort steht jedes Bit für ein Schutzattribut. Das erste Bit z.B. gibt an, ob die Datei gelöscht werden darf (Bit gelöscht) oder nicht (Bit gesetzt). In der Tabelle werden die Wertigkeiten der Bits angegeben. Wenn Sie mehrere Schutzattribute setzen möchten, können Sie die jeweiligen Werte aufaddieren:

Schutzattribut	Wert	Bedeutung
FIBF_DELETE	1	Löschschutz
FIBF_EXECUTE	2	Ausführungsschutz
FIBF_WRITE	4	Schreib-(Veränderungs-)schutz
FIBF_READ	8	Leseschutz
FIBF_ARCHIVE	16	Datei ist archiviert
FIBF_PURE	32	Datei ist resident ladbar
FIBF_SCRIPT	64	Datei ist eine Batch-Datei
FIBF_HIDE	128	Datei soll nicht angezeigt werden

FIBF_DELETE Die Datei kann nicht gelöscht werden.

FIBF_EXECUTE Die Datei kann nicht als Programm geladen und ausgeführt werden.

FIBF_WRITE Die Datei kann nicht verändert (beschrieben) werden.

FIBF_READ Die Datei kann nicht gelesen werden.

FIBF_ARCHIVE Dieses Bit dient zur Anzeige, daß die Datei archiviert wurde. Ein Backup-Programm (z.B. Festplatten-Backup) setzt das Bit nach der Sicherung. Bei jeder Veränderung an der Datei wird es automatisch vom DOS wieder gelöscht. Das Backup-Programm kann so feststellen, welche Dateien seit dem letzten Backup verändert wurden.

FIBF_PURE Das Pure-Bit zeigt an, daß die Datei, die ein Programm beinhalten sollte, mit dem CLI-Befehl 'Resident' fest in den Speicher geladen werden kann. Das Programm muß dazu sowohl mehrfach hintereinander gestartet als auch von mehreren Tasks gleichzeigtig benutzt werden können.

FIBF_SCRIPT Dieses Bit kennzeichnet die Datei als Script-Datei (ASCII-Datei mit CLI-Befehlen), die über den Execute-Befehl des CLI ausgeführt werden kann. Ab Kickstart 1.3 werden Dateien mit gesetztem Script-Bit bei alleiniger Eingabe ihres Namens per Execute ausgeführt, die Angabe von Execute ist nicht mehr nötig.

FIBF_HIDE Dieses Bit soll bewirken, daß die Datei im Inhaltsverzeichnis nicht mehr erscheint (wie die MS-DOS-Systemdateien), wird aber vom derzeitigen Betriebssystem noch nicht ausgewertet.

Als Beispiel setzen wir den Schutzstatus der Datei "df0:test" auf Löschbar, Lesbar, Schreibbar und Scriptdatei:

```
        move.l  #filename,d1
        move.l  #77,d2          ; = 1 + 4 + 8 + 64 (siehe Tabelle)
        jsr     -186(a6)
        ...

filename:     dc.b    "df0:test",0
              even
```

Bild 4.11: Setzen des Datei-Schutzstatus

4.4 Bearbeitung von Verzeichniseinträgen

Der nächste CLI-Befehl, dessen Funktionsweise wir untersuchen wollen, ist der Dir-Befehl. Wir beschäftigen uns also mit dem Einlesen und Bearbeiten von Verzeichnissen. Dazu erläutern wir Ihnen einige Grundlagen.

4.4.1 Datenstrukturen in Assembler

Ein sehr wichtiger Bestandteil des Amiga-Systems sind die sogenannten "Datenstrukturen" ("Structures"). Eine Struktur ist im Prinzip eine Ansammlung von hintereinander im Speicher stehenden Daten. Ähnlich, wie man die CPU-Register für Library-Aufrufe mit Daten füllt, kann man sich auch den Aufbau einer Struktur vorstellen: Angenommen, wir wollen die für eine Textausgabe wichtigen Informationen, die z.B. von einer Library-Routine benötigt werden, in einer Struktur zusammenstellen, dann sagen wir, ab der symbolischen Adresse (Label) 'struct' soll sie in unserem Programm beginnen.
Zuerst wollen wir jedoch die Vorder- und Hintergrundfarbe eintragen:

```
struct:     dc.b    1,0         ; Vorder- und Hintergrundfarbe
```

Damit haben wir zwei Bytes, jedes für eine Farbe, abgelegt. Ein Byte reicht hier aus, da die Farbnummer im Normalfall höchstens 64 sein kann. Im Gegensatz zu den Register-Belegungen bei Library-Aufrufen, wo man 'MOVE.L' benutzen kann, ist bei Strukturen die Wahl der richtigen Datengröße für die einzelnen Einträge sehr wichtig, da sich die weiteren Einträge ja sonst verschieben würden. Als nächstes soll die Startkoordinate des Textes in x- und y-Position folgen:

```
            dc.w    0,10        ; Startposition X und Y
```

Der Text wird also horizontal ab Pixel 0 (also ganz links am Rand) und vertikal ab Pixel 10 ausgegeben. Hier ist schon ein Wort nötig (x-Position reicht bis 640). Nun brauchen wir noch den Text selber. In unsere "Text-Info-Struktur" fügen wir einen Zeiger auf den Text, der woanders im Programm steht (bzw. stehen kann), ein:

```
            dc.l    text        ; Zeiger auf Text
```

Da Adressen immer "lang" sind, verwenden wir hier ein Langwort. Der Text selber kommt gleich hinter der Struktur, mit einem eigenen Label (das 'text', das wir in der Struktur verwendet haben):

```
text:       dc.b    "Test-Text",0
            even
```

Hier sehen Sie schon die Methode, Zeiger (z.B. auf andere Strukturen oder auch Texte und Daten) in Strukturen zu verwenden. Nochmal die ganze Struktur:

```
struct:     dc.b    1,0             ; Farben
            dc.w    0,10            ; X- und Y-Startposition
            dc.l    text            ; Zeiger auf Text

text:       dc.b    "Test-Text",0
            even
```

Wichtig ist, daß der Text mit dem Label davor, nicht mehr zur Struktur selber gehört.

Nun könnten wir z.B. einer Library-Routine einen Zeiger auf die Struktur übergeben und von ihr den Text ausgeben lassen. Das könnte so aussehen:

```
        lea     struct,a0       ; Zeiger auf Struktur-Start
        jsr     Print(a6)       ; Fiktive Library-Routine
```

Beachten Sie, daß es eine Library-Routine "Print", die mit unserer Beispiel-Struktur etwas anfangen könnte, nicht gibt. In der Intuition-Library gibt es aber eine Routine zur Textausgabe, die eine ähnliche, etwas größere Struktur erwartet.

4.4.2 BCPL-Pointer und BCPL-Strings

BCPL ist eine C-ähnliche Programmiersprache, und die BCPL-Pointer und -Strings sind zwei Datentypen dieser Sprache. BCPL hat die Besonderheit, den Speicher nicht byte-orientiert, sondern langwort-orientiert anzusprechen. Man kann sich das in Assembler übertragen so vorstellen, daß alle Befehle dort mit '.L' enden müssen.

Ein BCPL-Pointer ist, wie ein Assembler-Pointer, eine Speicheradresse, an der Daten beginnen. Der Unterschied ist der, daß ein BCPL-Pointer immer die Adresse dividiert durch 4 enthält. Um aus einem BCPL-Zeiger die Adresse zu erhalten, muß man ihn also mit 4 malnehmen. Daraus folgt auch die Beschränkung auf Langwort-Adressen: Wenn man eine Zahl mit 4 malnimmt, bekommt man immer eine Adresse, die genau auf einer Langwort-Grenze liegt.

Ein BCPL-String ist ein Text im Speicher, der in seinem ersten Byte die Länge enthält. Der eigentliche Text beginnt ab dem zweiten Byte. Der BCPL-String muß, wie der BCPL-Pointer, an einer Langwort-Grenze beginnen. Wenn ein BCPL-String in einer Struktur enthalten ist, bedeutet das, daß in der Struktur ein BCPL-Zeiger auf den eigentlichen String, der woanders im Speicher liegen kann, steht.

Für uns ist im Umgang mit BCPL-Strukturen wichtig, dafür zu sorgen, daß sie an einer Langwort-Grenze liegen. Beim Devpac verwendet man dazu die Direktive "cnop 0,4". Sie hat eine

ähnliche Wirkung wie die "even"-Direktive, sie richtet das Programm aber nicht nur auf eine gerade, sondern auf eine durch 4 teilbare Adresse aus. Also genau das, was wir brauchen. Hinweis: Beim Profimat-Assembler heißt diese Direktive 'align.l'!

4.4.3 Die File-Info-Block-Struktur

Zum Einbau von eigenen Strukturen in Programme werden wir später noch ausführlich kommen. Im Moment wollen wir keine Routinen mit Daten versorgen, wir wollen Daten von den Routinen bekommen. Viele Library-Routinen legen ihre Informationen an das aufrufende Programm auch in einer Struktur ab und schicken als Rückgabewert einen Zeiger auf sie. Dies tut auch die Routine, die zur Untersuchung von Verzeichniseinträgen zuständig ist. Zur Routine selbst kommen wir gleich, erst wollen wir uns die Struktur, in der wir unsere Infos bekommen, begutachten. Sie nennt sich File-Info-Block-Struktur (kurz FIB) und sieht so aus:

Die FileInfoBlock-Struktur

```
000   dc.l   fib_DiskKey          ; Nummer des Verwaltungsblocks
004   dc.l   fib_DirEntryType     ; > 0 bei Verz., sonst Datei
008   ds.b   fib_FileName,108     ; Dateiname, max. 30 Zeichen
116   dc.l   fib_Protection       ; Schutzstatus
120   dc.l   fib_EntryType
124   dc.l   fib_Size             ; Dateilänge in Bytes
128   dc.l   fib_NumBlocks
132   ds.b   fib_DateStamp,12     ; Zeitpunkt der letzten Änderung
144   ds.b   fib_Comment,116      ; Dateikommentar
260          fib_SIZEOF           ; SIZEOF = Größe der Struktur
```

Von nun an werden alle Strukturen, die uns über den Weg laufen, in dieser Form vorgestellt. Jede Zeile steht für einen Eintrag. Ganz links steht jeweils der (dezimale) Offset, das heißt der Abstand des Eintrages von der Startadresse der Struktur in Bytes. Dann folgt der Eintrags-Typ. Ein 'dc' steht für einen einzelnen Wert als Byte, Wort oder Langwort. Das 'ds.b' steht für einen Eintrag mit mehr als 4 Bytes (wie Name, Kommentar usw). In der letzten Zeile der Struktur ist dieser Platz leer. Achtung: diese Zeile ist kein Eintrag der Struktur mehr, sondern gibt nur ihre Länge in Bytes an. Als nächstes kommt die Commodore-Bezeichnung des Eintrages (so zu finden in den Include-Files). Im Falle von 'ds.b' kommt hinter der Bezeichnung die Größe des Eintrages, durch ein Komma abgetrennt und als Dezimalzahl. In der letzten Zeile steht hier "_SIZEOF", was den Eintrag für die Strukturgröße deutlich machen soll. Hinter dem Semikolon folgt eine kurze Erklärung. Eine ausführliche Erläuterung aller Einträge bzw. all derer, die im aktuellen Programm-Zusammenhang benötigt werden, erfolgt nach der Struktur, und zwar auf folgende Weise:

fib_DiskKey
Dieser Eintrag gibt die Blocknummer auf dem Datenträger an, auf dem die Verwaltungsdaten des Files oder Directories zu finden sind. Damit werden wir uns näher befassen, wenn wir Diskettenaufbau und File-System besprechen.

fib_DirEntryType
Hier steht eine 0, falls es sich um eine Datei handelt, bei einem Verzeichnis ein Wert ungleich 0.

fib_FileName
Obwohl dieser Eintrag 108 Bytes in der Struktur einnimmt, darf der Dateiname höchstens 30 Zeichen lang sein.

fib_Protection
Dieses Langwort entspricht der Schutz-Maske, wie sie bei der SetProtection-Routine verwendet wird. Nähere Erklärungen siehe dort.

fib_Size
Dürfte wohl klar sein.

fib_DateStamp
Dieser Eintrag enthält den Zeitpunkt der letzten Änderung an der Datei, wobei Datum und Uhrzeit in einer etwas ungewöhnlichen Kodierung angegeben werden. Der Eintrag besteht aus drei Langworten, wobei das erste die Anzahl der seit dem 1.1.1978 vergangenen Tage enthält, das zweite die Anzahl Minuten seit Mitternacht und das dritte die Anzahl der 50stel Sekunden in der laufenden Minute. Dummerweise stellt das DOS keine Umrechnungs-Routine zur Verfügung, wir müssen diese also selbst schreiben.

fib_Comment
Auch klar.

Zur Untersuchung von Verzeichnissen und Dateien gibt es zwei Routinen: Examine und ExNext (Untersuche und Untersuche nächsten Eintrag). Sie legen die gerade besprochene Struktur an, wir müssen ihnen nur sagen, wo in den Speicher sie geschrieben werden soll, sprich wir müssen Platz im Programm für die Struktur reservieren. Sie ist laut SIZEOF $104 Bytes lang, also verwenden wir im Programm folgenden Befehl:

```
            cnop    0,4
fib:        ds.b    $104
```

Damit haben wir Platz im Speicher. Wie Sie sehen, ist die FIB-Struktur eine BCPL-Struktur, weshalb die 'cnop'-Direktive immer vor der Speicherreservierung für einen FIB stehen muß. Die Examine-Routine sieht folgendermaßen aus:

Examine	=	**-102 (DOS-Library)**

***lock**	**d1**	<	Lock des zu untersuchenden Objekts
***fib**	**d2**	<	Zeiger auf Speicherstelle, an der FIB angelegt wird
success	**d0**	>	0 = Fehler aufgetreten
Erklärung			Untersucht eine Datei oder ein Verzeichnis

Mit Examine kann man sowohl Dateien als auch Verzeichnisse untersuchen. Der Eintrag fib_DirEntryType in der Struktur wird dann entsprechend gesetzt. Klar ist auch, daß der Eintrag für die Dateilänge bei einem Verzeichnis unbenutzt ist. Nach dem Examine-Aufruf können wir die gewünschten Informationen aus der FIB-Struktur ablesen. Ein Beispiel: Die Länge der Datei "df0:test" soll ermittelt werden:

```
         move.l   #filename,d1    ; Lock auf Datei holen
         move.l   #-2
         jsr      -84(a6)

         move.l   d0,d1           ; Lock nach d1
         move.l   #fib,d2         ; Startadresse unseres FIB-Speichers
         jsr      -102(a6)        ; Examine aufrufen
         tst.l    d0              ; Fehler?
         beq      fehler          ; Wenn ja

         lea      fib,a0          ; Startadresse FIB
         move.l   $7c(a0),d0      ; Die Länge hat den Offset $7c
         ...

filename:     dc.b    "df0:test",0
              even
              cnop    0,4
fib:          ds.b    $104
```

Bild 4.12: Auslesen der Dateilänge aus der FIB-Struktur

Danach steht die Dateilänge in d0. So weit, so gut. Nur wollten wir uns ja nicht eine einzelne Datei anschauen, sondern wir wollten wissen, welche Dateien denn in einem Verzeichnis stehen. Dazu müssen wir zunächst mal das Verzeichnis "examinen" (Lock holen, dann Examine aufrufen). Haben wir das getan, stehen die Informationen über das Verzeichnis in unserem FIB. Mit der Routine ExNext können wir uns jetzt immer den nächsten Eintrag des Verzeichnisses geben lassen. Die Informationen kommen in den selben FIB, den wir zu Beginn für das Verzeichnis verwendet hatten. ExNext sieht so aus:

ExNext	=	**-108 (DOS-Library)**

*lock	d1	<	Lock auf das zu untersuchende Verzeichnis
*fib	d2	<	Zeiger auf FIB-Speicherbereich
success	d0	>	0 = Keine weiteren Dateien
Erklärung			Untersucht den nächsten Eintrag eines Verzeichnisses

Die ExNext-Routine wird also genauso aufgerufen wie die Examine-Routine. Der große Unterschied ist, daß Examine den FIB immer mit den Informationen des gelockten Objekts vollschreibt, ExNext aber versucht, den nächsten Eintrag eines mit Examine untersuchten Verzeichnisses zu holen. Auch ist wichtig, daß eine Verzeichnis-Durchsuchung immer zuerst mit Examine eingeleitet werden muß, bevor ExNext benutzt werden darf. Nach Examine auf ein Verzeichnis steht aber nicht sofort der erste Eintrag im FIB, sondern zuerst die Informationen des Verzeichnisses selbst. Den ersten Eintrag bekommen wir erst nach ExNext.

4.4.4 Verzeichnisausgabe mit Examine und ExNext

Als Beispiel nun eine Routine, die "df0:" einliest und nach jedem Eintrag in eine (hier nicht eingebaute) Bearbeitungs-Routine springt:

```
        move.l  dirname,d1      ; Lock auf "df0:" holen
        move.l  #-2,d2
        jsr     -84(a6)
        move.l  d0,d4           ; Lock sichern

        move.l  d4,d1           ; "df0:" examinen
        move.l  #fib,d2
        jsr     -102(a6)
        tst.l   d0              ; Fehler?
        beq     fehler          ; Wenn ja

lab1:   move.l  d4,d1           ; Nächsten Eintrag holen
        move.l  #fib,d2         ; Wieder FIB-Adresse
        jsr     -108(a6)        ; ExNext aufrufen
        tst.l   d0              ; Weiterer Eintrag vorhanden?
        beq     keineintr       ; Wenn nein

        lea     fib,a0          ; Startadresse des FIB
        move.l  4(a0),d0        ; Eintrags-Art nach d0
        lea     8(a0),a1        ; Startadresse Name nach a1
        move.l  $7c(a0),d1      ; Länge nach d1
        bsr     printeintr      ; Fiktive Routine "Eintrag ausgeben"
```

```
        bra     labl            ; Zur Hauptschleife
        ...

dirname:        dc.b    "df0:",0
                even
                cnop    0,4
fib:            ds.b    $108
```

Bild 4.13: Verzeichnisausgabe über FIB

Damit hätten wir die Grundlagen zum Durchforsten eines Verzeichnisses. Ein Komplett-Programm zu diesem Thema gibt's natürlich auch, und das wollen wir uns jetzt anschauen. Es soll die Einträge eines von uns bestimmbaren Verzeichnisses ausgeben. Hinter dem Namen soll "(DIR)" stehen, wenn es ein Unterverzeichnis ist und die Länge, wenn es eine Datei ist. Den Namen des zu lesenden Verzeichnisses holen wir diesmal zur Abwechslung nicht aus der Kommandozeile, sondern wir benutzen die Read-Routine über das Input-Handle.

```
* Programm 4.3: Einfacher Dir-Befehl

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
Output          =       -60
Write           =       -48
Input           =       -54
Read            =       -42
Lock            =       -84
Examine         =       -102
ExNext          =       -108
UnLock          =       -90

        move.l  ExecBase,a6     ; DOS-Lib öffnen
        lea     dosname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,a6

        jsr     Output(a6)      ; Output- und Input-Handle holen
        move.l  d0,d4
        jsr     Input(a6)
        move.l  d0,d5

        lea     text1,a0        ; Aufforderung zur Eingabe des Ver-
        bsr     print           ; zeichnisnamens

        move.l  d5,d1           ; Verzeichnisname von Tastatur lesen
        move.l  #dirname,d2
        move.l  #50,d3
        jsr     Read(a6)
```

```
        lea       dirname,a0        ; Return-Zeichen nach Dir-Name durch
        move.b    #0,-1(a0,d0)      ; Null-Byte ersetzen

* Lock auf Verzeichnis holen

        move.l    #dirname,d1       ; Name des Dir
        move.l    #-2,d2            ; Lock-Typ SHARED
        jsr       Lock(a6)          ; Lock holen
        tst.l     d0                ; Fehler?
        bne       main1             ; Wenn nein

        lea       text2,a0          ; Fehler-Text
        bsr       print             ; Ausgeben
        bra       ende              ; Zum Programmende

* Verzeichnis examinen

main1:  move.l    d0,d6             ; Lock sichern

        move.l    d6,d1             ; Lock nach d1
        move.l    #fib,d2           ; Startadresse FIB nach d2
        jsr       Examine(a6)       ; Examine aufrufen

* Nächsten Eintrag holen

main2:  move.l    d6,d1             ; Lock nach d1
        move.l    #fib,d2           ; FIB nach d2
        jsr       ExNext(a6)        ; Nächsten Eintrag holen
        tst.l     d0                ; Noch Einträge da?
        beq       main4             ; Wenn nein

* Informationen ausgeben

        move.l    #fib+8,a0         ; Startadresse Dateiname im FIB
        bsr       print             ; Name ausgeben

        tst.l     fib+4             ; Eintragstyp = Verzeichnis?
        blt       main3             ; Wenn nein (kleiner Null=File)

        lea       text3,a0          ; Text " (DIR)" ausgeben
        bsr       print
        bra       main2

main3:  move.l    fib+$7c,d0        ; Dateilänge aus FIB nach d0
        lea       dezbuff,a0        ; Buffer für Dezimal-ASCII-String
        bsr       dezascii          ; Umwandlungsroutine aufrufen

        lea       text4,a0          ; Text " (" mit Länge dahinter ausgeben
        bsr       print
        lea       text5,a0          ; Text " Bytes)" ausgeben
        bsr       print
        bra       main2             ; Zur Hauptschleife
```

```
* Verzeichnis-Ende, Dir UnLocken

main4:  move.l   d6,d1            ; Lock nach d1
        jsr      UnLock(a6)       ; UnLocken

ende:   move.l   a6,a1            ; Lib schließen und Ende
        move.l   ExecBase,a6
        jsr      CloseLib(a6)

        rts

print:  movem.l  d1-d3,-(sp)      ; SUB Textausgabe für DOS-Write
        move.l   a0,d2            ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l    d3               ;  d4 < Handle der Ausgabedatei
pr1:    addq     #1,d3
        tst.b    (a0)+
        bne      pr1
        subq     #1,d3

        move.l   d4,d1
        jsr      Write(a6)
        movem.l  (sp)+,d1-d3
        rts

dezascii:                         ; SUB Umrechnung Dez-Zahl -> ASCII-Text
        movem.l  d1/d2/a1,-(sp)   ;  d0 < Dezimalzahl
        clr.b    d2               ; *a0 < Pufferzeiger
        lea      values,a1

da1:    clr.b    d1

da2:    addq     #1,d1
        sub.l    (a1),d0
        bcc      da2
        add.l    (a1),d0
        subq     #1,d1

        tst.b    d1
        beq      da3

        add.b    #"0",d1
        move.b   d1,(a0)+
        moveq    #1,d2
        bra      da4

da3:    tst.b    d2
        beq      da4
        move.b   #"0",(a0)+

da4:    tst.b    d0
        beq      da5

        add.l    #4,a1
        bra      da1

da5:    move.b   #0,(a0)
```

```
        movem.l  (sp)+,d1/d2/a1
        rts

* Datenbereich

dosname:        dc.b    "dos.library",0
                even
text1:          dc.b    "Name des auszugebenden Verzeichnisses: ",0
                even
text2:          dc.b    "Fehler beim Zugriff auf Verzeichnis!",10,0
                even
dirname:        ds.b    50
text3:          dc.b    " (DIR)",10,0
                even
text4:          dc.b    " ("
dezbuff:        ds.b    12
text5:          dc.b    " Bytes)",10,0
                cnop    0,4
fib:            ds.b    $104
values:         dc.l    1000000000,100000000,10000000
                dc.l    1000000,100000,10000,1000,100,10,1
```

Programm 4.3: Einfacher Dir-Befehl

Das war schon ein recht langes Programm, nicht wahr? In den Zeilen

```
    lea       dirname,a0       ; Return-Zeichen nach Dir-Name durch
    move.b    #0,-1(a0,d       move.b    #0,-1(a0,d0)    ; Null-Byte ersetzen
```

wird das Return-Zeichen am Ende des eingegebenen Textes durch ein Null-Byte ersetzt. Diese Methode haben wir schon einmal angewandt (beim CD-Programm).

Wenn beim Locken des Verzeichnisses ein Fehler auftritt, wird in

```
        lea       text2,a0         ; Fehler-Text
        bsr       print            ; Ausgeben
        bra       ende             ; Zum Programmende
```

eine entsprechende Meldung ausgegeben. Auf einen Fehler-Test nach dem Examine können wir verzichten (wenn Lock geklappt hat, wird Examine bestimmt auch klappen). Bei ExNext müssen wir einen eventuellen Fehler allerdings berücksichtigen, denn als solcher zählt ja auch der Fall "kein Verzeichnis-Eintrag mehr da".

Im Datenbereich haben wir den Text Nr. 4 unmittelbar vor dem Dez-ASCII-Wandel-Puffer stehen, ohne abschließendes Null-Byte:

```
text4:          dc.b    " ("
dezbuff:        ds.b    12
```

Damit erreichen wir, daß wir den Text und die umgerechnete Zahl in einem Print-Aufruf ausgeben können, da die Dez-ASCII-Routine ja ihrerseits ein Ende-Nullbyte hinter die Zahl schreibt. Beachten Sie auch das 'cnop 0,4' vor dem FIB-Puffer (BCPL-Struktur)!

4.4.5 Disketten-Informationen

Nachdem wir nun den Inhalt eines Verzeichnisses ausgelesen haben, wollen wir noch ein paar Informationen über die Diskette selbst haben (Diskette bezieht sich hier nicht nur auf Floppy-Disks, sondern auch auf Festplatten, die RAM-Disk u.ä.) Wir wollen ihren Namen, ihre Größe und ihren Füllungsgrad wissen. Dazu müssen wir uns zunächst mit einer weiteren Struktur herumschlagen, der InfoData-Struktur:

Die InfoData-Struktur

```
00    dc.l   id_NumSoftErrors      ; Anzahl "weicher" Fehler
04    dc.l   id_UnitNumber         ; Nummer des Laufwerks
08    dc.l   id_DiskState          ; Disketten-Zustand, siehe unten
12    dc.l   id_NumBlocks          ; Anzahl Datenblöcke
16    dc.l   id_NumBlocksUsed      ; Anzahl belegter Blöcke
20    dc.l   id_BytesPerBlock      ; Anzahl Bytes in einem Block
24    dc.l   id_DiskType           ; Disketten-Typ, siehe unten
28    dc.l   id_VolumeNode
32    dc.l   id_InUse              ; 1 = Disk wird benutzt
36           id_SIZEOF
```

id_NumSoftErrors
Hier wird die Anzahl der seit Systemstart auf der Diskette aufgetretenen "weichen" Fehler (Checksummenfehler, falsche Blocktypen u.ä.) eingetragen.

id_UnitNumber
Wenn man nicht die Laufwerksbezeichnung (DF0-DF3) zur Ansprache einer Disk benutzt, sondern ihren Namen, ist es vielleicht interessant zu wissen, in welchem Laufwerk die Disk liegt.

id_DiskState
Folgende Werte sind möglich:

Diskstatus	Wert	Bedeutung
ID_WRITE_PROTECTED	80	Schreibgeschützt
ID_VALIDATING	81	Wird gerade auf Gültigkeit überprüft
ID_VALIDATED	82	Ist gültig

ID_WRITE_PROTECTED Dürfte wohl klar sein.

ID_VALIDATING Die Diskette wurde gerade eingelegt und wird nun vom DOS auf Gültigkeit überprüft. Solange dieser Zustand anhält, kann von der Disk nur gelesen werden.

ID_VALIDATED	Wenn die Disk gültig ist, wechselt der Status auf VALIDATED. Von jetzt an kann auch auf die Disk geschrieben werden. Wenn sie fehlerhaft ist, bleibt der Status auf VALIDATING.

id_NumBlocks
Die Anzahl der insgesamt auf der Disk vorhandenen Datenblöcke. Beim Amiga-System umfaßt ein Datenblock immer 512 Bytes. Eine normale Diskette mit 880 KB hat 1760 Blöcke.

id_NumBlockUsed
Die Anzahl der benutzten Blöcke, die unbenutzten kann man über die Differenz zu NumBlocks berechnen.

id_BytesPerBlock
Anzahl verfügbarer Datenbytes in einem Block. Von den 512 Bytes in jedem Block werden 24 für Verwaltungsinformationen verwendet. Daher sind zur reinen Datenspeicherung 488 Bytes in jedem Block verfügbar.

id_DiskType
Folgende Werte sind möglich:

Disk-Typ	Wert	Bedeutung
ID_NO_DISK_PRESENT	-1	Keine Disk im Laufwerk
ID_UNREADABLE_DISK	$42414400	Disk nicht lesbar
ID_NOT_REALLY_DOS	$4E444F53	Keine DOS-Disk
ID_DOS_DISK	$444F5300	Disk o.k.
ID_FFS_DISK	$444F5301	FastFileSystem-Disk
ID_KICKSTART_DISK	$4B49434B	A1000-Kickstart-Disk

ID_NO_DISK_PRESENT	Klar, oder?
ID_UNREADABLE_DISK	Die Diskette ist nicht lesbar. Die equ-Langworte dieses und der folgenden drei Werte stellen jeweils vier ASCII-Zeichen, die ja ein Byte lang sind, dar. Bei UNREADABLE ist es "BAD" (und ein 0-Byte).
ID_NOT_REALLY_DOS	Die Diskette ist lesbar, ihr fehlen aber wichtige Informationen für das Amiga-DOS. Die vier ASCII-Zeichen sind "NDOS".
ID_DOS_DISK	Eine ganz normale DOS-Diskette ("DOS"+0-Byte).
ID_FFS_DISK	Eine Diskette, die mit dem FastFileSystem (schnelleres Dateisystem vor allem für Festplatten, ab Kickstart 2.0 auch für Disketten) formatiert wurde. Kennung: "DOS"+1-Byte.
ID_KICKSTART_DISK	Die Diskette, die beim Amiga 1000 das Betriebssystem enthält ("KICK"). Der Amiga 1000 hat, im Gegensatz zu den sonstigen Amigas, sein Betriebssystem nicht im ROM, sondern auf einer Diskette. Nur die grundlegendsten Routinen zum Laden von der Disk stehen im ROM.

Wie Sie sehen, steht in dieser Struktur alles drin, was wir benötigen. Wichtig: Auch dies ist eine BCPL-Struktur (cnop 0,4 nicht vergessen)! Gefüllt wird sie von der DOS-Routine Info:

Info	=	**-114 (DOS-Library)**

***lock**	**d1**	<	Lock auf irgendein Objekt auf der Diskette
***parameter**	**d2**	<	Adresse des Speichers für InfoData-Struktur
success	**d0**	>	0 = Fehler aufgetreten
Erklärung			Holt Informationen über eine Diskette

Wir müssen diese Routine mit einem Lock, das sich in irgendeiner Form auf die gewünschte Diskette bezieht, versorgen. Am besten nimmt man die Geräte-Bezeichnung (z.B. "df0:"). Dazu ein kurzes Beispiel:

```
        move.l  #diskname,d1    ; Geräte-Bezeichnung
        move.l  #-2,d2          ; Lock-Typ SHARED
        jsr     -84(a6)         ; Lock holen

        move.l  d0,d1           ; Lock nach d1 für Info
        move.l  #dinfo,d2       ; Speicher für Struktur
        jsr     -114(a6)        ; Info holen
        ...

diskname:   dc.b    "df0:",0
            even
            cnop    0,4             ; Nicht vergessen!
dinfo:      ds.b    $24
```

Bild 4.14: InfoData-Struktur füllen

Und gleich noch ein Komplett-Programm hinterher. Den Parameter (Laufwerksbezeichnung) geben wir diesmal wieder in der Kommandozeile an. Wir wollen den Namen wissen (zu finden im FileName-Eintrag des FIB, der bei Examine auf ein Laufwerk den Diskettennamen enthält), die Gesamtblockzahl und die Anzahl der freien Blöcke (in der InfoData-Struktur). Also frisch ans Werk:

```
* Programm 4.4: Ausgabe von Disketten-Informationen

ExecBase    =       4
OpenLib     =       -552
CloseLib    =       -414
```

```
Output          =       -60
Write           =       -48
Lock            =       -84
Info            =       -114
Examine         =       -102
UnLock          =       -90

        movem.l a0/d0,-(sp)     ; Kommandozeile sichern

        move.l  ExecBase,a6     ; DOS-Lib öffnen
        lea     dosname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,a6

        jsr     Output(a6)      ; Output-Handle holen
        move.l  d0,d4

        movem.l (sp)+,a5/d5     ; Kommandozeile zurück
        move.b  #0,-1(a5,d5)    ; Schluß-Return durch Nullbyte ersetzen

        move.l  a5,d1           ; Lock auf Diskette holen
        move.l  #-2,d2
        jsr     Lock(a6)
        tst.l   d0              ; Fehler?
        bne     main1           ; Wenn nein

        lea     text1,a0        ; Fehler-Text
        bsr     print           ; ausgeben
        bra     ende            ; Zum Programmende

* InfoData-Struktur füllen

main1:  move.l  d0,d6           ; Lock sichern
        lea     dinfo,a4        ; Speicher für DiskInfo in Merk-Register
        lea     fib,a5          ; FIB-Speicher ebenso
        move.l  d6,d1           ; Lock nach d1
        move.l  a4,d2           ; DiskInfo-Speicher nach d2
        jsr     Info(a6)        ; Info holen

* Diskettentyp prüfen

        cmp.l   #$444f5300,$18(a4)      ; DOS-Diskette?
        beq     main2                   ; Wenn ja
        cmp.l   #$444f5301,$18(a4)      ; FastFileSystem-Diskette?
        beq     main2                   ; Wenn ja

* Keine DOS-Diskette im Laufwerk

        lea     text2,a0        ; Fehlertext
        bsr     print           ; ausgeben
        bra     main4           ; Zum UnLocken

* Diskettennamen ausgeben

main2:  move.l  d6,d1           ; Lock
```

```
        move.l   a5,d2          ; Speicher für FIB
        jsr      Examine(a6)    ; Examine rufen
        lea      text3,a0       ; Text "Diskname: " ausgeben
        bsr      print
        lea      8(a5),a0       ; Disknamen ausgeben
        bsr      print

        cmp.l    #81,8(a4)      ; Diskstatus "Validating"?
        bne      main3          ; Wenn nein

* Diskette wird validiert, Block-Infos sind ungültig

        lea      text4,a0       ; Fehler-Text
        bsr      print          ; ausgeben
        bra      main4

main3:  lea      dezbuff,a3     ; Speicher für Dezzahl in Merkregister

* Anzahl Datenblöcke ausgeben

        lea      text5,a0       ; Text "Anzahl Datenblöcke: "
        bsr      print
        move.l   12(a4),d0      ; Anzahl Blöcke nach d0
        move.l   a3,a0          ; Dez-Puffer
        bsr      dezascii       ; Zahl umrechnen
        move.l   a3,a0          ; Pufferinhalt
        bsr      print          ; ausgeben

* Anzahl freier Blöcke und Bytes ausgeben

        lea      text6,a0       ; Text "Freie Blöcke"
        bsr      print
        move.l   12(a4),d3      ; Gesamtzahl Blöcke
        sub.l    16(a4),d3      ; Belegte Blöcke abziehen
        move.l   d3,d0          ; Nach d0
        move.l   a3,a0          ; Dez-Puffer
        bsr      dezascii       ; Umrechnen
        move.l   a3,a0          ; Puffer
        bsr      print          ; ausgeben

        lea      text7,a0       ; Text "Freie Bytes"
        bsr      print
        mulu     #488,d3        ; Anzahl Blöcke mal 488 (siehe Anm.)
        move.l   d3,d0          ; Umrechnen
        move.l   a3,a0
        bsr      dezascii
        move.l   a3,a0          ; Puffer
        bsr      print          ; ausgeben

        lea      text8,a0       ; Text " Bytes)" und Return-Zeichen
        bsr      print

main4:  move.l   d6,d1          ; Lock auf Diskette
        jsr      UnLock(a6)     ; UnLocken

ende:   move.l   a6,a1          ; Lib schließen
```

```
        move.l   ExecBase,a6
        jsr      CloseLib(a6)
        rts                          ; Und tschüß!

print:  movem.l  d1-d3,-(sp)        ; SUB Textausgabe für DOS-Write
        move.l   a0,d2              ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l    d3                 ;  d4 < Handle der Ausgabedatei
pr1:    addq     #1,d3
        tst.b    (a0)+
        bne      pr1
        subq     #1,d3

        move.l   d4,d1
        jsr      Write(a6)
        movem.l  (sp)+,d1-d3
        rts

dezascii:                           ; SUB Umrechnung Dez-Zahl -> ASCII-Text
        movem.l  d1/d2/a1,-(sp)     ;  d0 < Dezimalzahl
        clr.b    d2                 ; *a0 < Pufferzeiger
        lea      values,a1

da1:    clr.b    d1

da2:    addq     #1,d1
        sub.l    (a1),d0
        bcc      da2
        add.l    (a1),d0
        subq     #1,d1

        tst.b    d1
        beq      da3

        add.b    #"0",d1
        move.b   d1,(a0)+
        moveq    #1,d2
        bra      da4

da3:    tst.b    d2
        beq      da4
        move.b   #"0",(a0)+

da4:    cmp.l    #1,(a1)
        beq      da5

        add.l    #4,a1
        bra      da1

da5:    move.b   #0,(a0)
        movem.l  (sp)+,d1/d2/a1
        rts


* Datenbereich

dosname:      dc.b    "dos.library",0
```

```
               even
text1:         dc.b    "Fehler beim Zugriff auf Laufwerk!",10,0
               even
text2:         dc.b    "Keine gültige DOS-Disk im Laufwerk!",10,0
               even
text3:         dc.b    "Diskettenname: ",0
               even
text4:         dc.b    10,"Disk nicht validiert - Anzahl freier "
               dc.b    "Blöcke ungültig!",10,0
               even
text5:         dc.b    10,"Anzahl Datenblöcke: ",0
               even
text6:         dc.b    10,"Anzahl freie Datenblöcke: ",0
text7:         dc.b    " (",0
               even
text8:         dc.b    " freie Bytes)",10,0
               even
               cnop    0,4
fib:           ds.b    $104
dinfo:         ds.b    $24
dezbuff:       ds.b    12
values:        dc.l    1000000000,100000000,10000000
               dc.l    1000000,100000,10000,1000,100,10,1
```

Programm 4.4: Ausgabe von Disketten-Informationen

Keine Sorge, die nächsten Programm werden wieder etwas kürzer ausfallen. Das erste Neue in diesem Programm ist das Füllen der Info-Struktur. Vor dem Info-Aufruf holen wir uns mit

```
        lea     dinfo,a4        ; Speicher für DiskInfo in Merk-Register
        lea     fib,a5          ; FIB-Speicher ebenso
```

die Startadressen unserer DiskInfo- und FIB-Struktur in Adreßregister, weil wir sie noch des öfteren brauchen werden und ein Zugriff auf ein Adreßregister schneller geht als jedesmal die Speicheradresse zu holen. Der Aufruf von Info mit

```
        move.l  d6,d1           ; Lock nach d1
        move.l  a4,d2           ; DiskInfo-Speicher nach d2
        jsr     Info(a6)        ; Info holen
```

wurde ja schon besprochen. Als nächstes wird der Disktyp geprüft. Im Falle von UNREADABLE oder NOT_DOS usw. wird ein Fehlertext ausgegeben. Zulässig sind die Typen DOS_DISK ($444f5300) und FFS_DISK ($444f5301). In diesen beiden Fällen geht es weiter im Text (bei 'main2'). Die Diskette wird examined. Dabei muß man wissen, daß bei Examine auf ein Verzeichnis der Verzeichnisname im FIB steht, und ebenso der Diskettenname bei Examine auf das Hauptverzeichnis. So kann der Name also direkt aus dem FileName-Eintrag ausgegeben werden.

Als nächstes wird getestet, ob der Diskstatus "Validating" ist, ob die Diskette also gerade die DOS-Gültigkeitsprüfung über sich ergehen lassen muß. In diesem Fall sind die Angaben über Blockanzahl und belegte Blöcke noch nicht gültig, der entsprechende Ausgabeteil im Programm wird also übersprungen. Ansonsten wird die Blockzahl aus der DiskInfo-Struktur ausgelesen, umgerechnet und ausgegeben:

```
move.l  12(a4),d0       ; Anzahl Blöcke nach d0
move.l  a3,a0           ; Dez-Puffer
bsr     dezascii        ; Zahl umrechnen
move.l  a3,a0           ; Pufferinhalt
bsr     print           ; ausgeben
```

Die Anzahl freier Blöcke wird berechnet, indem von der Gesamtblockzahl die Anzahl belegter Blöcke abgezogen wird. Diese Zahl wird ausgegeben:

```
move.l  12(a4),d3       ; Gesamtzahl Blöcke
sub.l   16(a4),d3       ; Belegte Blöcke abziehen
move.l  d3,d0           ; Nach d0
move.l  a3,a0           ; Dez-Puffer
bsr     dezascii        ; Umrechnen
```

Dann wird der freie Diskplatz noch in Bytes ausgegeben. Die Blockanzahl wird mit 488 malgenommen. Eigentlich sind zwar 512 Bytes in jedem Diskblock, aber aus systemtechnischen Gründen nutzt das Amiga-DOS nur 488 Bytes jedes Blocks für reine Daten. Beim FastFileSystem ist das allerdings nicht mehr so.

Der Rest des Programms (Disk UnLocken, Lib schließen) dürfte klar sein, ebenso die schon bekannten Unterroutinen 'print' und 'dezascii'.

4.5 Arbeit mit einfachen DOS-Fenstern

Wir verlassen nun den Bereich der Verzeichnisbearbeitung und kommen zu den DOS-Spezialgeräten.

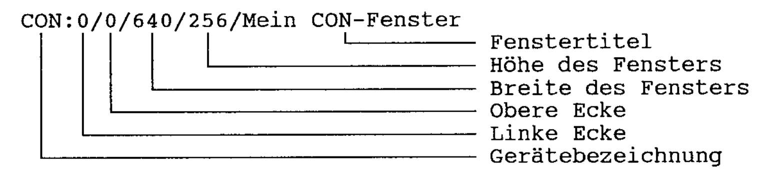
Wir wir schon wissen, sind der Begriff "Datei" und die damit verbundenen Routinen nicht streng an eine Disketten-Datei gebunden. Es gibt im DOS die Möglichkeit, den Drucker, die serielle Schnittstelle oder das Sprachausgabesystem anzusprechen, und man kann sogar einfache Fenster öffnen. Das ist auch unser nächstes Thema: die DOS-Fenster.

Die Gerätebezeichnungen "DF0:" bis "DF3:" für die Diskettenlaufwerke sind sicherlich bekannt. Neben diesen gibt es aber noch einige weitere, z.B. für eine Festplatte ("DH0:"), für die RAM-Disk ("RAM:"), die resetfeste RAM-Disk ("RAD:") usw. Die DOS-Fenster werden auch über solche Gerätenamen verwaltet, nämlich über "CON:" und "RAW:". Zunächst wollen wir uns

mit CON: beschäftigen. Die mit CON: geöffneten Fenster haben sowohl Vor- als auch Nachteile gegenüber "richtigen" Intuition-Fenstern (mit denen wir uns im nächsten Kapitel beschäftigen werden). Sie sind wesentlich einfacher zu handhaben als Intuition-Fenster, bieten aber nicht so umfangreiche Möglichkeiten, ihr Aussehen und ihre Funktionen zu verändern.

4.5.1 Öffnen und Schließen von CON-Fenstern

CON: steht für "Console", also die Tastatur. Dieses Bezeichnung soll demnach eigentlich die Tastatur ansprechen, aber da man normalerweise auch sehen will, was man eintippt, wird ein Fenster geöffnet, das die Eingaben zeigt.
So wie man beim Ansprechen des Disklaufwerks hinter dem Gerätenamen den Dateinamen angibt, schreibt man bei der Benutzung von CON: diverse Informationen über das zu öffnende Fenster hinter den Gerätenamen. Genauer gesagt, man gibt an, wo die linke, obere Ecke des Fensters liegen soll, wie groß es sein soll und welchen Titel es bekommen soll. Diese Parameter trennt man jeweils durch einen Schrägstrich ab. Gleich ein Beispiel:



Diese Angaben für CON würden also ein Fenster öffnen, dessen linke, obere Ecke ganz oben auf dem Bildschirm liegt, das 640 Punkte breit und 256 Punkte hoch ist (also die volle Bildschirmgröße einnimmt) und dessen Fenstertitel "Mein CON-Fenster" lautet. Da die DOS-Spezialgeräte wie Dateien behandelt werden, kann man den ganzen Text als Dateinamen ansehen. Um das Fenster zu öffnen, brauchen wir also lediglich die DOS-Routine Open mit diesem Namen aufzurufen, und die Close-Routine, um es wieder zu schließen:

```
        move.l  #conname,d1     ; Zeiger auf Fenstertext
        move.l  #1005,d2        ; Status = Gerät schon vorhanden
        jsr     -30(a6)         ; Open aufrufen
        move.l  d0,d5           ; Handle sichern
        ...
        move.l  d5,d1           ; Handle nach d1
        jsr     -36(a6)         ; Fenster schließen per Close
        ...

conname:        dc.b    "CON:0/0/640/256/Mein CON-Fenster",0
                even
```

Bild 4.15: CON-Fenster öffnen und schließen

4.5.2 Lesen und Schreiben von CON-Fenstern

Ebenso wie zum Öffnen und Schließen verwendet man auch zur Ein- und Ausgabe in CON-Fenster die normalen DOS-Dateiroutinen. Natürlich sind nicht alle Routinen sinnvoll anwendbar, wie sollte man z.B. ein CON-Fenster löschen oder umbenennen oder es als Verzeichnis ausgeben können. Neben Open und Close sind nur Read und Write für CON-Dateien zugelassen. Mehr brauchen wir auch gar nicht.

Die Ein- und Ausgabe in ein CON-Fenster funktioniert genauso wie mit den Input- und Output-Handles (also dem Standard-CLI-Fenster), hier wird allerdings das selbe Handle für Ein- und Ausgabe benutzt. Ein Beispiel darf natürlich nicht fehlen:

```
* Programm 4.5: Ein- und Ausgabe in ein CON-Fenster

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
Open            =       -30
Write           =       -48
Read            =       -42
Close           =       -36

        move.l   ExecBase,a6       ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

* CON-Fenster öffnen

        move.l   #conname,d1       ; CON-Dateiname nach d1
        move.l   #1005,d2          ; Gerät schon vorhanden
        jsr      Open(a6)          ; 'Datei' öffnen
        tst.l    d0                ; Fehler beim Öffnen?
        beq      ende              ; Wenn ja
        move.l   d0,d4             ; Handle in d4 sichern

        lea      text1,a0          ; Text im Fenster ausgeben
        bsr      print

* Text von Tastatur lesen

        move.l   d4,d1             ; CON-Handle
        move.l   #buff,d2          ; Startadresse Textbuffer
        move.l   #40,d3            ; Max. 40 Zeichen
        jsr      Read(a6)          ; Von CON lesen
        move.l   d0,d3             ; Länge sichern
```

```
* Gelesenen Text wieder ausgeben:

        lea     text2,a0        ; Einleitungs-Text
        bsr     print

        lea     buff,a0         ; Pufferstart
        move.b  #0,0(a0,d0)     ; Text mit Nullbyte abschließen
        bsr     print           ; Und raus damit

* Auf Return-Tastendruck warten (1 Zeichen einlesen)

        move.l  d4,d1           ; CON-Handle
        move.l  #buff,d2        ; Pufferstart
        move.l  #1,d3           ; Eingabetext ist unwichtig
        jsr     Read(a6)        ; Lesen

* CON-Fenster schließen

        move.l  d4,d1           ; Handle
        jsr     Close(a6)       ; 'Datei' schließen

ende:   move.l  a6,a1           ; Lib schließen und Auf Wiedersehen
        move.l  ExecBase,a6
        jsr     CloseLib(a6)
        rts

print:  movem.l d1-d3,-(sp)     ; SUB Textausgabe für DOS-Write
        move.l  a0,d2           ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l   d3              ;  d4 < Handle der Ausgabedatei
prl:    addq    #1,d3
        tst.b   (a0)+
        bne     prl
        subq    #1,d3

        move.l  d4,d1
        jsr     Write(a6)
        movem.l (sp)+,d1-d3
        rts

* Datenbereich

dosname:        dc.b    "dos.library",0
                even
conname:        dc.b    "con:0/0/640/60/Ein CON-Fenster",0
                even
text1:          dc.b    "Geben Sie einen Text ein: ",0
                even
text2:          dc.b    "Sie haben eingegeben: ",0
                even
buff:           ds.b    40
```

Programm 4.5: Ein- und Ausgabe in ein CON-Fenster

Ob es sinnvoll ist, einen Text einzulesen und sofort wieder auszugeben, sei dahingestellt. Am besten betrachten Sie nur den Demonstrationszweck des Programms. Neues bringt es eigentlich nicht, es soll nur noch einmal zeigen, daß man auf ein CON-Fenster genauso zugreifen kann wie auf eine Datei.

4.5.3 RAW-Fenster contra CON-Fenster

Das zweite Fenster-Konsolen-Gerät des DOS nennt sich "RAW:". Das bedeutet soviel wie "roh", woraus sich schließen läßt, daß dieser Fenstertyp dem Anwender nicht so viel Arbeit abnimmt wie CON. Letzteres übernimmt nämlich schon alle Editierfunktionen: Es zeigt den eingetippten Text im Fenster an und läßt Korrekturen mit den Backspace-Taste zu. Dafür hat es aber auch einen Nachteil: Nicht alle Tastendrücke werden ans aufrufenden Programm weitergeleitet. Das gilt z.B. für die Backspace-Taste, aber auch für die ESC-, DEL- und HELP-Taste oder die Funktionstasten. Wenn man diese Tasten im Programm auswerten will, z.B. in einem Menü, kann man mit CON nichts anfangen.

RAW verhält sich etwas anders: Es zeigt den eingetippten Text weder selbstständig im Fenster an, noch läßt es Korrekturen mit Backspace o.ä. zu. Alle Tastendrücke werden so wie sie kommen (eben 'roh') ans aufrufende Programm weitergeleitet. Das gilt auch für die Funktionstasten, die man bei Benutzung von RAW problemlos abfragen kann. Der Nachteil von RAW liegt allerdings auf der Hand: Man kann im Programm nicht einfach sagen: Lese 40 Zeichen von RAW an die Adresse "buff". Im Prinzip wäre Texteingabe auf diese Weise zwar möglich, aber der Benutzer würde dann ja gar nicht sehen, was er tippt und hätte keinerlei Möglichkeiten zur Korrektur.

Bei Benutzung von RAW darf man sich also immer nur ein Zeichen von der Read-Routine geben lassen, welches dann im Programm selbst ausgewertet werden muß. Im einfachsten Fall muß also ein Backspace in Form von Löschen der letzten Eingabe (falls schon etwas eingegeben wurde) im Speicher und auf dem Bildschirm bearbeitet werden, Return zeigt das Eingabeende and und sonstige (gültige) Zeichen müssen auf dem Bildschirm ausgegeben und in den Textspeicher geschrieben werden. Wir müssen also einiges an Arbeit auf uns nehmen, die CON uns abnimmt, dafür können wir aber unseren eigenen, individuellen Editor schreiben.

4.6 Sonstige DOS-Geräte

Nachdem wir uns nun ausgiebig mit CON- und RAW-Fenstern beschäftigt haben, wollen wir sehen, was es sonst noch für DOS-Geräte gibt. Wichtig zu wissen ist, daß es zwei Arten von Geräten gibt, die richtigen Geräte (DF0:, CON: usw.) und

die sog. logischen Geräte. Diese stehen nicht für ein wirkliches Gerät wie ein Disklaufwerk oder die Tastatur, sondern sind frei wählbare Bezeichnungen, die einem Verzeichnis oder einer Datei auf einem Datenträger zugewiesen werden. Beispiele dafür sind die LIBS:, FONTS: usw. Diese werden beim Systemstart den entsprechenden Verzeichnissen auf der Startdiskette zugewiesen. Ansehen und ändern kann man diese Zuweisungen mit dem CLI-Befehl 'Assign'. Weitere Informationen zum Thema logische Geräte finden Sie im Anhang "CLI-Schnellkurs".

Eine Zusammenstellung aller Gerätebezeichnungen, die im Amiga-System verwendet werden, finden Sie ebenfalls im CLI-Anhang.

4.6.1Benutzung von NIL: in Assembler

Wie CON: und RAW: kann auch NIL: als Datei geöffnet werden. Als Dateiname braucht nur NIL: angegeben zu werden. Das Gerät verwirft alle Daten, die an es gesendet werden, und gibt bei Leseversuchen immer 'wirklich gelesene Bytes = 0' zurück. Interessant ist NIL: im Zusammenhang mit der DOS-Execute-Funktion.

4.6.2Das verbesserte Konsolen-Gerät NEWCON:

NEWCON: ist nur in Kickstart 1.3 verfügbar. In früheren Kickstart-Versionen existiert es noch nicht, und ab Kickstart 2.0 hat das normale Konsolen-Gerät CON die selben Funktionen wie NEWCON. Um NEWCON unter 1.3 benutzen zu können, muß es gemounted werden. Der "newcon-handler" im L-Verzeichnis und ein entsprechender Eintrag in der Mountlist sind also nötig.

Die Benutzung von NEWCON entspricht der von CON, bietet allerdings einige verbesserte Funktionen zur Editierung. Dies sind folgende:

1. Mit den Cursortasten Links und Rechts kann im eingegebenen Text herumgewandert werden.
2. Die DEL-Taste entfernt das Zeichen unter dem Cursor.
3. Mit Cursor-Hoch-Runter kann man die Liste der bisher eingegebenen Texte durchblättern. In der Eingabezeile erscheint dann jeweils der vorige (Cursor hoch) oder nächste (Cursor runter) Befehl.

Wenn Sie Kickstart 1.3-Besitzer sind und die Möglichkeit haben, NEWCON zu benutzen, sollten Sie diese auf jeden Fall wahrnehmen, da NEWCON wesentlich umgänglicher ist als CON.

4.6.3 Schnittstellenansprache mit SER:, PAR: und PRT:

Beim Öffnen der entsprechenden Datei, die normalerweise nur beschrieben werden kann (es sei denn, Sie haben ein Modem o.ä. am seriellen Port), muß nur die Gerätebezeichnung als Dateiname angegeben werden. Das Schreiben erfolgt genauso wie bei einer "normalen" Datei.

Das folgende Beispielprogramm gibt die in der Kommandozeile angegebene Diskdatei auf dem PRT:-Drucker aus (bei der Datei sollte es sich natürlich günstigerweise um eine Textdatei handeln). Die Dateigröße ist auf 5000 Zeichen begrenzt, Sie können das aber auch durch Vergrößerung des Puffers im Datenbereich ändern (falls Sie wirklich von diesem Beispielprogramm eine Riesen-Textdatei ausdrucken lassen wollen).

```
* Programm 4.6: Ausgabe einer Datei auf dem Drucker

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
Open            =       -30
Output          =       -60
Write           =       -48
Read            =       -42
Close           =       -36

        movem.l a0/d0,-(sp)

        move.l  ExecBase,a6     ; DOS-Lib öffnen
        lea     dosname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,a6

        jsr     Output(a6)      ; Output-Handle holen
        move.l  d0,d4

        movem.l (sp)+,a0/d0
        move.l  #0,-1(a0,d0)    ; Return durch Nullbyte ersetzen

* Inhalt der Textdatei einlesen

        move.l  a0,d1           ; Kommandozeile = Dateiname
        move.l  #1005,d2        ; Datei existiert (hoffentlich)
        jsr     Open(a6)        ; Öffnen
        tst.l   d0              ; Fehler beim Öffnen?
        bne     main1           ; Wenn nein

        lea     text1,a0        ; Fehler beim Dateizugriff
        bsr     print           ; Ausgeben
        bra     ende
```

```
main1:  move.l   d0,d5          ; Handle sichern

        move.l   d5,d1          ; Textdatei
        move.l   #buff,d2       ; Startadresse Textbuffer
        move.l   #5000,d3       ; Max. 5000 Zeichen
        jsr      Read(a6)       ; Einlesen
        move.l   d0,d6          ; Länge sichern

        move.l   d5,d1          ; Textdatei
        jsr      Close(a6)      ; Schließen

* Drucker-Datei öffnen

        move.l   #prtname,d1    ; Dateiname für Drucker
        move.l   #1005,d2       ; Datei existiert (hoffentlich)
        jsr      Open(a6)       ; Öffnen
        tst.l    d0             ; Fehler beim Öffnen?
        bne      main2          ; Wenn ja

        lea      text2,a0       ; Fehler beim Öffnen des Druckers
        bsr      print          ; Ausgeben
        bra      ende

main2:  move.l   d0,d5

        move.l   d5,d1          ; Handle des Druckers
        move.l   #buff,d2       ; Pufferstart
        move.l   d6,d3          ; Länge
        jsr      Write(a6)      ; Gut druck!

* Drucker-Datei schließen

        move.l   d5,d1          ; Drucker-Handle
        jsr      Close(a6)      ; schließen

ende:   move.l   a6,a1          ; Lib schließen und Ende
        move.l   ExecBase,a6
        jsr      CloseLib(a6)
        rts

print:  movem.l  d1-d3,-(sp)    ; SUB Textausgabe für DOS-Write
        move.l   a0,d2          ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l    d3             ;  d4 < Handle der Ausgabedatei
pr1:    addq     #1,d3
        tst.b    (a0)+
        bne      pr1
        subq     #1,d3

        move.l   d4,d1
        jsr      Write(a6)
        movem.l  (sp)+,d1-d3
        rts

* Datenbereich

dosname:     dc.b    "dos.library",0
```

```
         even
text1:   dc.b    "Fehler beim Öffnen der Datei!",10,0
         even
text2:   dc.b    "Fehler beim Öffnen des Druckers!",10,0
         even
prtname: dc.b    "PRT:",0
         even
buff:    ds.b    5000
```

Programm 4.6: Ausgabe einer Datei auf dem Drucker

4.6.4 Sprachausgabe in Assembler mit SPEAK:

Die Ansteuerung dieses Gerätes erfolgt genauso wie bei SER:, PAR: und PRT:. Sie können also das vorige Beispielprogramm geringfügig ändern (das "PRT:" in der drittletzten Zeile in ein "SPEAK:" ändern), um die Textdatei nicht auf dem Drucker auszugeben, sondern sich "vorlesen" zu lassen. Viel Spaß!

4.7 Ausführung von CLI-Befehlen

Nun kommen wir zu einer weiteren interessanten Möglichkeit, die uns die DOS-Library bietet: Die Ausführung der CLI-Befehle von Programmen aus. Dazu dient die DOS-Routine Execute, nicht zu verwechseln mit dem gleichnamigen CLI-Befehl (obwohl die Arbeitsweise sehr ähnlich ist):

Execute		=	**-222 (DOS-Library)**
***string**	**d1**	<	Zeiger auf Kommandotext oder 0, falls nicht gewünscht
***infile**	**d2**	<	Handle einer Datei, aus der weitere CLI-Befehle gelesen werden oder 0, falls nicht gewünscht
***outfile**	**d3**	<	File-Handle einer Datei, in die eventuelle Ausgaben der Befehle geschickt werden (z.B. CON-Fenster)
status	**d0**	>	0 = Fehler aufgetreten
Erklärung			Führt CLI-Kommandos aus (einzelne Befehlszeile und/oder Stapeldatei)

In d1 übergibt man der Routine einen Zeiger auf den Text des auszuführenden Befehls. Dieser entspricht einer ganz normalen CLI-Eingabezeile mit Befehlsnamen und Kommandozeile. Da jeder CLI-Befehl einen Ausgabekanal (Datei, CON-Fenster o.ä.) braucht, an den er eventuelle Texte schicken kann, muß in d3 ein File-Handle übergeben

werden. Das kann z.B. auch das Handle sein, das man von der Output-Routine bekommt.

Besonders interessant ist die Möglichkeit, ein Handle auf eine Textdatei zu übergeben (das 'infile'), aus dem weitere CLI-Befehle gelesen werden. Jede Zeile dieser Datei, die mit einem normalen Return (ASCII-10) abgeschlossen sein muß, wird als eine Befehlszeile interpretiert. Eine solche Datei nennt man "Batchfile" ("Stapeldatei"). Da die Angabe der einzelnen Befehlszeile in d1 nicht nötig ist (d1 kann auch auf 0 stehen), dürfte Ihnen so langsam klar werden, wie der CLI-Execute-Befehl arbeitet ...

4.7.1 Ausführung von einzelnen Befehlen

Als erstes Beispiel-Programm wollen wir den CLI-Befehl "dir df0:" von ausführen lassen:

```
* Programm 4.7: Ausführung eines CLI-Befehls

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
Output          =       -60
Execute         =       -222

        move.l  ExecBase,a6     ; DOS-Lib öffnen
        lea     dosname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,a6

        jsr     Output(a6)      ; Output-Handle holen
        move.l  d0,d4

        move.l  #command,d1     ; Zeiger auf Kommandotext
        clr.l   d2              ; Keine Eingabedatei
        move.l  d4,d3           ; Ausgabe nach Standard-Output
        jsr     Execute(a6)     ; Execute aufrufen

        move.l  a6,a1           ; Lib schließen und Ende
        move.l  ExecBase,a6
        jsr     CloseLib(a6)
        rts

* Datenbereich

dosname:     dc.b    "dos.library",0
             even
command:     dc.b    "dir df0:",0
             even
```

Programm 4.7: Ausführung eines CLI-Befehls

Sie können natürlich auch ein anderes Kommando ausführen lassen, indem Sie die entsprechende Zeile im Datenbereich ändern. Beachten Sie, daß vor dem Execute-Aufruf mit

```
        move.l  d4,d3           ; Ausgabe nach Standard-Output
```

festgelegt wird, wohin Ausgaben des Befehls geschickt werden sollen. Das ist sehr wichtig. Auch bei Ausführung von CLI-Befehlen, die gar keine Ausgabe produzieren, muß ein Ausgabekanal angegeben werden, sonst gibt es einen Systemabsturz. Falls Sie Textausgabe unterdrücken wollen, können Sie natürlich auch eine Datei mit dem Namen "NIL:" öffnen und die Ausgabe dorthin schicken.

4.7.2 Ausführung von Stapeldateien

Nun wollen wir noch den Execute-Befehl des CLI nachprogrammieren. In der Kommandozeile wird der Name der auszuführenden Datei übergeben, die geöffnet wird. Das Handle der Datei wird an Execute weitergereicht.

```
* Programm 4.8: Ausführung von Stapeldateien

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
Open            =       -30
Output          =       -60
Write           =       -48
Execute         =       -222
Close           =       -36

        movem.l a0/d0,-(sp)

        move.l  ExecBase,a6     ; DOS-Lib öffnen
        lea     dosname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,a6

        jsr     Output(a6)      ; Output-Handle holen
        move.l  d0,d4

        movem.l (sp)+,a0/d0
        move.l  #0,-1(a0,d0)    ; Return durch Nullbyte ersetzen

* Datei öffnen

        move.l  a0,d1           ; Kommandozeile = Dateiname
        move.l  #1005,d2        ; Datei existiert (hoffentlich)
        jsr     Open(a6)        ; Öffnen
        tst.l   d0              ; Fehler beim Öffnen?
```

```
        bne       main1            ; Wenn nein

        lea       text1,a0         ; Fehler beim Dateizugriff
        bsr       print            ; Ausgeben
        bra       ende

main1:  move.l    d0,d5            ; Handle sichern

* Stapeldatei ausführen

        clr.l     d1               ; Keine einzelne Befehlszeile
        move.l    d5,d2            ; Input-File = geöffnete Datei
        move.l    d4,d3            ; Output-File = Standard-Output
        jsr       Execute(a6)      ; Datei ausführen

        move.l    d5,d1            ; Datei wieder schließen
        jsr       Close(a6)        ; (nicht vergessen!)

ende:   move.l    a6,a1            ; Lib schließen und Ende
        move.l    ExecBase,a6
        jsr       CloseLib(a6)

        rts

print:  movem.l   d1-d3,-(sp)      ; SUB Textausgabe für DOS-Write
        move.l    a0,d2            ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l     d3               ;  d4 < Handle der Ausgabedatei
pr1:    addq      #1,d3
        tst.b     (a0)+
        bne       pr1
        subq      #1,d3

        move.l    d4,d1
        jsr       Write(a6)
        movem.l   (sp)+,d1-d3
        rts

* Datenbereich

dosname:      dc.b    "dos.library",0
              even
text1:        dc.b    "Fehler beim Dateizugriff!",10,0
```

Programm 4.8: Ausführung von Stapeldateien

Die Verfahrensweise ist fast dieselbe wie beim letzten Programm. Hier wird allerdings beim Execute-Aufruf der Zeiger auf die einzelne Befehlszeile (d1) auf Null gesetzt und in d2 das Handle, der zuvor geöffneten Eingabedatei, eingetragen. Ansonsten ist alles bekannt.

4.8 Laden und Ausführen von Segmenten

Neben der Möglichkeit, einen Befehl (also ein Programm) per Execute auszuführen, wobei die Steuerung an das CLI übergeben wird, kann man die Ausführung auch "selbst in die Hand nehmen". Es gibt eine DOS-Routine, die eine Programmdatei in den Speicher lädt, die nötigen Anpassungen vornimmt (Umrechnung der absoluten Adressen) und dem aufrufenden Programm die Startadresse mitteilt.

Die Adreßumrechnung, die im Abschnitt über die PC-relative Adressierung (2.4.9) schon einmal erwähnt wurde, geschieht anhand einer Tabelle, die im Anschluß an das eigentliche Programm in der Datei steht. In dieser Tabelle sind alle Stellen im Programm (relativ zum Programmanfang) verzeichnet. Auf alle diese Adressen wird einfach die wirkliche Startadresse des Programms im Speicher aufaddiert.

Damit eine Datei als ausführbares Programm erkannt wird, muß sie einige Bedingungen erfüllen. So müssen z.B. die Adreß-Umrechnungstabelle und einige Verwaltungsdaten vorhanden sein. Im Normalfall brauchen wir uns um die Einrichtung der Tabelle und der sonstigen Daten nicht zu kümmern, das übernimmt der Assembler für uns. Den genauen Aufbau eines ausführbaren Programms werden wir in einem späteren Kapitel besprechen.

Die Programmlade-Routine des DOS heißt LoadSeg:

LoadSeg	=	**-150 (DOS-Library)**

***filename**	**d1**	>	Zeiger auf Namenstext des zu ladenden Programmes
***segment**	**d0**	>	BCPL-Zeiger auf erstes Segment oder 0 bei Fehler
Erklärung			Lädt ein Programm als Segment ein

Das 'Seg' steht für Segment. Ein Programm kann aus mehreren Teilen bestehen, die nicht unbedingt hintereinander im Speicher stehen müssen. Diese Teile nennt man Segmente. Die Laderoutine legt die Segmente entsprechend des verfügbaren Speicherplatzes ab. In d0 bekommen wir von LoadSeg einen BCPL-Zeiger auf das erste Segment in der Programmdatei, welches auch immer als erstes gestartet wird. Da es sich um einen BCPL-Zeiger handelt, müssen wir ihn mit 4 malnehmen, um die wirkliche Speicheradresse zu erhalten. Das erste Langwort jedes Segmentes ist wiederum ein BCPL-Zeiger auf das jeweils nächste Segment (im letzten Segment steht in diesem Langwort eine 0), welchen wir überspringen müssen. Die von LoadSeg gemeldete Zahl muß also mit 4 multipliziert,

und auf diese Zahl muß 4 aufaddiert werden. Die Adresse, die wir dann bekommen, können wir per JSR anspringen. Allerdings brauchen wir die, von LoadSeg gemeldete Zahl selbst später auch noch, weshalb sie erhalten bleiben muß. Im Programm sähe das dann so aus:

```
move.l   d0,d1          ; Die LoadSeg-Rückgabe wird noch
                        ; gebraucht
asl.l    #2,d1          ; Zahl mal 4
addq     #4,d1          ; Und plus 4
move.l   d1,a0          ; In ein Adreßregister
jsr      (a0)           ; Und anspringen
```

Zunächst müssen wir allerdings noch ein bißchen Vorarbeit leisten. Man darf nämlich erstens nicht davon ausgehen, daß sich ein LoadSeg-Programm an die Routinenaufruf-Konventionen hält und alle Register bis auf d0, d1, a0 und a1 rettet. Vor dem JSR-Aufruf müssen wir daher die Register selbst sichern, am besten mit dem Befehl

```
movem.l  d0-d7/a0-a6,-(sp)
```

Nach dem Aufruf holen wir sie mit

```
movem.l  (sp)+,d0-d7/a0-a6
```

zurück. Im Prinzip muß man natürlich nicht alle Register retten, sondern nur die, welche später noch benötigt werden. Aber da es vor allem bei längeren Programmen immer schwerer wird, den Überblick über die benötigten Registerinhalte zu behalten, sichert man am besten alle (lieber ein Register mehr retten als eins zu wenig).

Bei manchen Programmen besteht die Möglichkeit, diese mehrfach hintereinander aufzurufen, ohne sie neu laden zu müssen. Das funktioniert dann, wenn das Programm nicht während des Ablaufs irgendwelche Werte (oder auch Befehle) verändert, die beim Start nicht wieder in ihre ursprüngliche Form zurückgebracht werden. Es gibt nur eine Möglichkeit, herauszufinden, ob ein Programm mehrfach startbar ist: Sie müssen es ausprobieren. Wenn es nicht geht, werden Sie es sehr wahrscheinlich an einem Absturz, zumindest aber an einem fehlerhaften Programmablauf merken.

Die DOS-Routine, die ein Programm nach der Ausführung wieder aus dem Speicher entfernt (sprich den belegten Speicher freigibt), heißt UnLoadSeg:

UnLoadSeg	=	**-156 (DOS-Library)**

***segment** **d1** < BCPL-Zeiger auf Start des freizugebenden Segments

error **d0** > 0 = Fehler aufgetreten

Erklärung Gibt Speicher von geladenen Segmenten frei

In d1 ist hier exakt der Wert einzutragen, den uns LoadSeg gemeldet hat, ohne irgendwelche Umrechnungen. Der Aufruf von UnLoadSeg ist wichtig, da erst dann der, vom Programm benutzte Speicherplatz wieder freigegeben wird.

Das folgende Programm lädt ein Programm, dessen Name in der Kommandozeile steht, per LoadSeg ein, führt es aus und ruft dann UnLoadSeg auf.

```
* Programm 4.9: Laden und Ausführen eines Programms als Segment

ExecBase      =       4
OpenLib       =       -552
CloseLib      =       -414
LoadSeg       =       -150
UnLoadSeg     =       -156

        movem.l  a0/d0,-(sp)

        move.l   ExecBase,a6    ; DOS-Lib öffnen
        lea      dosname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,a6

        movem.l  (sp)+,a0/d0    ; Kommandozeile zurück und Abschluß-
        move.b   #0,-1(a0,d0)   ; Return durch Nullbyte ersetzen

* Programm per LoadSeg einladen

        move.l   a0,d1          ; Name des Programmes
        jsr      LoadSeg(a6)    ; LoadSeg aufrufen
        tst.l    d0             ; Fehler beim Laden?
        beq      ende           ; Wenn ja, zum Ende

* Einsprungadresse berechnen

        move.l   d0,d1          ; BCPL-Pointer nach d1
        asl.l    #2,d1          ; d1 mal 4
        addq     #4,d1          ; d1 plus 4
        move.l   d1,a0          ; d1 in ein Adreßregister
```

```
* Programm aufrufen

        movem.l  d0-d7/a0-a6,-(sp)     ; Alle Register sichern
        jsr      (a0)                  ; Programm aufrufen
        movem.l  (sp)+,d0-d7/a0-a6     ; Register zurück

        move.l   d0,d1          ; Von LoadSeg gemeldeter Wert
        jsr      UnLoadSeg(a6)  ; Segment entfernen

ende:   move.l   a6,a1          ; Lib schließen und Ende
        move.l   4,a6
        jsr      CloseLib(a6)

        rts

* Datenbereich

dosname:      dc.b    "dos.library",0
              even
```

Programm 4.9: Laden und Ausführen mit LoadSeg

Weitere Kommentare zu diesem Programm sind wohl nicht nötig.

4.8.1 Multitasking-gerechtes Warten

Nun soll noch eine DOS-Routine vorgestellt werden, die thematisch recht gut zum Starten von Programmen paßt. Der Amiga ist ja bekanntlich eine Multitasking-Maschine. Neben den Programmen, die wir schreiben und ablaufen lassen, spielt sich noch eine ganze Menge mehr ab. Steht man nun vor der Aufgabe eine Verzögerung in sein Programm einbauen zu müssen, sollten man dazu nicht einfach eine "leere" Schleife programmieren (wie das früher in BASIC oft üblich war), da man damit den übrigen Tasks im System wertvolle Rechenzeit wegnehmen würde. Stattdessen benutzt man die DOS-Routine Delay. Diese versetzt unser Programm in den "Wartezustand", wodurch wir den Multitasking-Ablauf nicht mehr stören. Nach Ablauf einer festgelegten Zeitspanne wacht unser Programm wieder auf und setzt seine Arbeit fort. Die Delay-Routine sieht so aus:

Delay	=	**-198 (DOS-Library)**

timeout	d1	<	Anzahl der Ticks (=1/50 Sekunde), die gewartet werden soll
Erklärung			Versetzt den Task für 'ticks' 50stel Sekunden in den Wartezustand

Was der Begriff "Wartezustand" systemtechnisch bedeutet, wird im Exec-Kapitel geklärt. Im Moment müssen wir nur wissen, wie wir Verzögerungen in unsere Programme einbauen, ohne andere Tasks zu stören.

Kapitel 5
Die Intuition-Library

Windows

Screens

Intuition-Grafikstrukturen

Gadgets

Menüs

Requester

Auswertung von Nachrichten

Die Basis-Struktur der Intuition-Library

Nachdem wir die DOS-Library kennengelernt haben, wollen wir uns nun die **Intuition-Library** genauer ansehen. Sie bildet die Schnittstelle zwischen dem Anwender und dem Rechner. Mit ihren Funktionen kann man leicht eine benutzerfreundliche Oberfläche programmieren, die über Screens, Windows, Requester, Menüs und Gadgets verfügt.

5.1 Windows

Anfangen wollen wir mit den Windows (Fenstern). Im vorangegangenen Kapitel haben wir uns schon mit ihnen beschäftigt und sie wie Dateien mittels der Open-Anweisung der DOS-Library geöffnet. Wenn wir die OpenWindow-Funktion der Intuition-Library benutzen, müssen wir zwar auf die einfache Handhabung verzichten, dafür können wir aber auf das Aussehen stärker Einfluß nehmen.

OpenWindow	=	**-204 (Intuition-Library)**

***NewWindow**	**a0**	<	Zeiger auf eine initialisierte NewWindow-Struktur.
***Window**	**d0**	>	Zeiger auf die, von Intuition angelegte Window-Struktur des geöffneten Fensters oder eine Null, wenn das Fenster nicht geöffnet werden konnte.
Erklärung			öffnet ein Fenster, welches durch die Einträge der angegebenen Window-Struktur beschrieben ist. Außerdem wird eine neue Struktur angelegt, mit der das Fenster verwaltet wird.

Wie man sieht, benötigt die OpenWindow-Funktion nur einen Parameter. Dies ist ein Zeiger auf eine NewWindow-Struktur, die von uns zuvor angelegt werden muß. Anhand der eingetragenen Werte wird dann das Fenster geöffnet und eine neue Struktur, die Window-Struktur, von Intuition aufgebaut. Auf diese Struktur erhalten wir, nach Beendigung der Funktion, einen Zeiger zurück, mit dem wir unser Fenster ab jetzt ansprechen können.

Wenn wir schon ein Fenster öffnen können, ist es auch wichtig zu wissen, wie man es wieder schließt. Dazu gibt es die Funktion CloseWindow. Sie benötigt auch nur einen Parameter, und zwar den Zeiger auf die Window-Struktur, den wir von OpenWindow erhalten haben.

CloseWindow	=	**-72 (Intuition-Library)**

***Window** a0 < Zeiger auf die Window-Struktur, die man von OpenWindow erhalten hat.

Erklärung CloseWindow schließt das Fenster mit der angegebenen Window-Struktur.

Damit wären die wichtigsten Funktionen für Windows erklärt. Was uns nun noch fehlt ist die NewWindow-Struktur, die Intuition als Bauanleitung für unser Fenster dienen soll. Sie setzt sich wie folgt zusammen:

NewWindow-Struktur:

```
00    dc.w    nw_LeftEdge       ; linke Ecke
02    dc.w    nw_TopEdge        ; obere Ecke
04    dc.w    nw_Width          ; Breite
06    dc.w    nw_Height         ; Höhe
08    dc.b    nw_DetailPenl     ; Vordergrundfarbe
09    dc.b    nw_BlockPen       ; Hintergrundfarbe
10    dc.l    nw_IDCMPFlags     ; IDCMP-Flags
14    dc.l    nw_Flag           ; Flags
18    dc.l   *nw_FirstGadget    ; Zeiger auf erstes Gadget
22    dc.l   *nw_CheckMark      ; Grafik für Menühaken
26    dc.l   *nw_Title          ; Name des Festers
30    dc.l   *nw_Screen         ; Zeiger auf Screen
34    dc.l   *nw_BitMap         ; Zeiger auf eigene BitMap
38    dc.w    nw_MinWidth       ; X-Minimum des Fensters
40    dc.w    nw_MinHeight      ; Y-Minimum des Fensters
42    dc.w    nw_MaxWidth       ; X-Maximum des Fensters
44    dc.w    nw_MaxHeight      ; Y-Maximum des Fensters
46    dc.w    nw_Type           ; Art des Screens, auf dem das
                                ; Window erscheint
48            nw_SIZEOF
```

nw_LeftEdge, nw_TopEdge,nw_ Width, nw_Height
Positionierung und Größenangabe des Fensters.

nw_DetailPen, nw_BlockPen
Farbtabellennummer des Vorder- (DetailPen) und Hintergrundes (BlockPen).

nw_IDCMPFlags
Durch die Intuition Direct Communications Message Ports-Flags wird Intuition mitgeteilt, welche Ereignisse, die unser Fenster betreffen, an unser Programm (bzw. useren Message-Port) weiterzuleiten sind. Auf die Möglichkeiten, die uns damit zur Verfügung stehen, gehen wir etwas später ein. Die Flags haben folgende Bedeutung:

IDCMP-Flag	Wert	Bedeutung
SIZEVERIFY	$00000001	Größe des Fensters soll verändert werden

NEWSIZE	$00000002	Größe des Fensters wurde verändert
REFRESHWINDOW	$00000004	Fenster wurde überlagert
ACTIVEWINDOW	$00040000	Fenster wurde aktiviert
INACTIVEWINDOW	$00080000	Fenster wurde deaktiviert
GADGETDOWN	$00000020	GADGIMMADIATE-Gadget wurde gewählt
GADGETUP	$00000040	RELVERIFY-Gadget wurde angewählt
CLOSEWINDOW	$00000200	Close-Gadget wurde angewählt
REQSET	$00000080	Erster Requester wurde geöffnet
REQCLEAR	$00001000	Letzter Requester wurde geschlossen
REQVERIFY	$00000800	Requester soll geöffnet werden
MENUPICK	$00000100	Menüpunkt wurde gewählt
MENUVERIFY	$00002000	Menü soll gezeigt werden
MOUSEBUTTONS	$00000008	Eine Maustaste wurde gedrückt
MOUSEMOVE	$00000010	Maus wurde bewegt
DELTAMOVE	$00100000	Mausbewegung relativ
INTUITICKS	$00400000	Jede zehntel Sekunde ein Nachricht
NEWPREFS	$00004000	Preferences wurden geändert
DISKINSERTED	$00008000	Diskette wurde eingelegt
DISKREMOVED	$00010000	Diskette wurde entnommen
RAWKEY	$00000400	Tastatureingabe mit RAW-Codes
VANILLAKEY	$00200000	Eingabe mit bearbeiteten KeyCodes
WBENCHMESSAGE	$00020000	Nachricht von der WBench
LONELYMESSAGE	$80000000	Keine IDCMP-Nachricht

ACTIVEWINDOW Fenster wurde aktiviert.

INACTIVEWINDOW Fenster wurde deaktiviert.

GADGETDOWN Gadget des Typs GADGIMMEDIATE wurde betätigt.

GADGETUP Gadget des Typs RELVERIFY wurde betätigt.

CLOSEWINDOW CloseGadget des Fensters wurde angewählt. Achtung: Das Fenster wird nicht automatisch geschlossen!

SIZEVERIFY Benutzer versucht, das Fenster in seiner Ausdehnung zu verändern. (Um die Veränderung des Fensters zu gestatten, muß die Nachricht bestätigt werden. Erst dann wird die Größe geändert!)

NEWSIZE Die Größe des Fensters wurde geändert.

REFRESHWINDOW Bei Überlagerungen durch andere Fenster kann es passieren, daß der Inhalt unseres Fensters verlorengeht. Durch REFRESHWINDOW erfahren wir, wann ein Teil des Fensters verdeckt wurde und können dann den Inhalt wiederherstellen.

REQSET Der erste Requester des Fensters wurde erstellt.

REQCLEAR Der letzte Requester des Fensters wurde geschlossen.

REQVERIFY Intuition meldet, daß ein Requester geöffnet werden soll. Nach Bestätigung des Empfangs wird er erstellt.

MENUPICK Ein Menüpunkt ist ausgewählt worden. Dabei kann es auch sein, daß die rechte Maustaste (RMT) nur kurz gedrückt und wieder losgelassen wurde. Bei der

	Menüabfrage werden wir dieses Problem lösen.
MENUVERIFY	Intuition wartet auf Bestätigung der Meldung, um die Menüs wieder neu zu zeichnen.
MOUSEBUTTONS	Linke oder rechte Maustaste ist gedrückt worden. Sollte die linke Maustaste über der HITBOX eines Gadgets gedrückt worden sein, erhalten wir keine Nachricht. Außerdem bekommen wir für die rechte Maustaste nur eine Meldung, wenn das Flag RMBTRAP des Eintrags nw_Flag gesetzt ist!
MOUSEMOVE	Mausbewegung wird gemeldet (Relativ zur linken oberen Fensterecke)
DELTAMOVE	Mausbewegung wird gemeldet (Relativ zur letzten Position der Maus)
INTUITICKS	Eine zehntel Sekunde ist abgelaufen. Da nach einiger Zeit die Liste der eingegangenen Nachrichten mit INTUITICKS-Nachrichten überlaufen wäre, sendet Intuition nur eine weitere Meldung, wenn die vorangegangene mittels ReplyMsg quittiert wurde.
NEWPREFS	Die Preferences-Werte wurden geändert.
DISKINSERTED	Diskette wurde eingelegt.
DISKREMOVED	Diskette wurde entnommen.
RAWKEY	Eine Taste wurde gedrückt. Man erhält in der Message-Struktur den RAW-Code, also den unbehandelten Tastencode, zurück.
VANILLAKEY	Eine Taste wurde gedrückt. Man erhält, im Unterschied zu RAWKEY, den mit der Tastaturtabelle behandelten Wert zurück.
WBENCHMESSAGE	WorkBench-Message wurde empfangen.
LONELYMESSAGE	Keine Intuition-Message angekommen.

nw_Flags

Der Wert Flags enthält die Einstellung, die das Aussehen des Fensters festlegen. Folgende Flags sind definiert:

Windowflag	Wert	Bedeutung
SIZEBRIGHT	$00000010	Size-Gadget im rechten Rand
SIZEBOTTOM	$00000020	Size-Gadget im unteren Rand
WINDOWSIZING	$00000001	Gadget für Größeneinstellung
WINDOWDRAG	$00000002	Fenster kann verschoben werden
WINDOWDEPTH	$00000004	DEPTH-Gadget wird eingebunden
WINDOWCLOSE	$00000008	Close-Gadget wird eingebunden
BACKDROP	$00000100	Fenster direkt auf Screen legen
GIMMEZEROZERO	$00000400	Getrennte Verwaltung Inhalt & Rand
BORDERLESS	$00000800	Fenster ohne Ränder darstellen
ACTIVATE	$00001000	Fenster wird beim Öffnen aktiv
REPORTMOUSE	$00000200	Mauskoordinaten werden gemeldet
RMBTRAP	$00010000	Right**M**ouse**B**utton**TRAP** (kein Menü)
NOCAREREFRESH	$00020000	Keine Meldung wenn Fenster beschädigt

SIMPLE_REFRESH	$00000040	Fensterinhalt wird nicht erneuert
SMART_REFRESH	$00000000	verdeckte Bereiche sichern
SUPER_BITMAP	$00000080	Restauration aus eigener BitMap möglich

SIZEBRIGHT Das Size-Gadget soll im rechten Rand eingerichtet werden.

SIZEBOTTOM Das Size-Gadget soll im unteren Rand eingerichtet werden.

WINDOWSIZING Das Size-Gadget wird in die Window-Struktur eingebunden, damit das Fenster vom Benutzer beliebig in seiner Größe verändert werden kann.

WINDOWDRAG Das Fenster kann mit Hilfe der Maus an seiner Titelzeile auf dem Screen verschoaben werden.

WINDOWDEPTH Depth-Gadgets einbinden. Sie dienen dazu, das Fenster in den Vorder- oder Hintergrund zu bringen.

WINDOWCLOSE Das Close-Gadget wird in die Titelzeile des Windows eingebunden.

BACKDROP Das Fenster wird direkt auf dem Screen dargestellt. Es ist das unterste Fenster und kann nicht nach vorne geholt werden. System-Gadgets sind nicht erlaubt.

GIMMEZEROZERO Der Rand und der Inhalt des Fensters werden getrennt voneinander verwaltet.

BORDERLESS Das Fenster wird ohne Rahmen gezeichnet.

ACTIVATE Das Fenster wird direkt, nachdem es geöffnet wurde, aktiviert.

REPORTMOUSE Die Position der Maus wird durchgehend gemeldet.

RMBTRAP Das RightMouseButton-TRAP-Flag verhindert, daß die Menüzeile bei Druck auf die rechte Maustaste erscheint.

NOCAREREFRESH Es wird keine Meldung bei zerstörtem Fensterinhalt gesendet.

SIMPLE_REFRESH Die Erneuerung des Fensters wird nicht von Intuition übernommen. Diese Aufgabe muß vom Programm erledigt werden.

SMART_REFRESH Verdeckte Bereiche des Fensters werden zur späteren Wiederherstellung des Inhalts gespeichert.

SUPER_BITMAP Der gesamte Inhalt des Fensters wird in einer BitMap gespeichert, die bei Überlagerungsschäden zur Regenerierung des Fensters genutzt werden.

***nw_FirstGadget**
Zeiger auf die Struktur des ersten Gadget, welches eingebunden werden soll (Null, wenn kein Gadget eingebunden werden soll).

***nw_CheckMark**

Adresse der Grafikdaten, welche verwendet werden sollen, um ein Menüpunkt als ausgewählt zu markieren (Null, wenn der Standardhaken benutzt werden soll).

***nw_Title**
Zeiger auf eine Zeichenkette, die den Text der Titelzeile des Fensters enthält. Der Text muß mit einem Null-Byte abgeschlossen werden.

***nw_Screen**
Zeiger auf die Screen-Struktur, auf der das Fenster erstellt werden soll. Soll das Fenster auf dem WorkBench-Screen gebildet werden, setzt man eine Null ein.

***nw_BitMap**
Adresse einer eigenen, schon initialisierten BitMap-Struktur, die an Stelle der von Intuition erstellten, benutzt werden soll. Soll keine eigene benutzt werden, setzt man eine Null ein.

nw_MinWidth, nw_MinHeight, nw_MaxWidth, nw_MaxHeight
Minimale und maximale Breite und Höhe, die das Fenster annehmen kann.

nw_Type
Zuletzt bleibt nur noch der Eintrag Type, dessen Wert angibt, ob das Fenster auf dem Workbench-Screen oder einem eigenen (CUSTOMSCREEN) erscheinen soll.

```
WBENCHSCREEN        =         1
CUSTOMSCREEN        =         15
```

Jetzt haben wir alle wichtigen Schritte kennengelernt, die zum öffnen eines Fensters nötig sind. Deshalb wollen wir nicht länger zögern und unser erstes eigenes Intuition-Window basteln.

```
* Programm 5.1: Öffnen und Schließen eines Windows

ExecBase     =      4
CloseLib     =      -414
OldOpenLib   =      -408
OpenWindow   =      -204
CloseWindow  =      -72

Start:
        move.l  ExecBase,a6     ; ExecBase nach a6
        lea     IntName,a1      ; Zeiger auf Intui-Namen
        jsr     OldOpenLib(a6)  ; Library öffnen
        move.l  d0,IntBase      ; Basisadresse speichern
        beq     IntError        ; Fehler aufgetreten ?
```

```
        move.l  IntBase,a6      ; BasisAdresse laden
        lea     WindowArgs,a0   ; Windowdef. in a0 übergeben
        jsr     OpenWindow(a6)  ; Fenster öffnen
        move.l  d0,WindowHD     ; Zeiger auf Window speichern
        beq     WinError        ; Fehler aufgetreten ?

WaitLMT:
        btst    #6,$bfe001      ; linke Maustaste gedrückt ?
        bne     WaitLMT         ; nein, dann warten

        move.l  IntBase,a6      ;
        move.l  WindowHD,a0     ; Zeiger auf das Fenster
        jsr     CloseWindow(a6)         ; übergeben und schließen

WinError:
        move.l  ExecBase,a6     ;
        move.l  IntBase,a1      ; Intuition-Library schließen
        jsr     CloseLib(a6)    ;

IntError:
        rts                     ; ReTurn from Subroutine


* Datenbereich

IntBase:        dc.l    0               ; Speicher für Int-Base
WindowHD:       dc.l    0               ; Speicher WindowPointer

IntName:        dc.b    "intuition.library",0
                        ; Name der Library, die geöffnet werden soll
                even    ; PC auf geraden Wert bringen

WinName:        dc.b    "LMT drücken, um Programm zu beenden!",0
                even

WindowArgs:                             ; Die NewWindow-Struktur
                dc.w    90,10           ; Position des Fensters
                dc.w    460,200         ; Breite und Höhe
                dc.b    1,3             ; Farbe Hintergrund/Vordergrund
                dc.l    $600            ; IDCMP-Flags
                dc.l    $1100F          ; Window-Flags
                dc.l    0               ; Zeiger auf erstes Gadget
                dc.l    0               ; Zeiger auf CheckMark Grafik
                dc.l    WinName         ; Zeiger auf Titelzeichenkette
ScreenHD:       dc.l    0               ; Zeiger auf einen Screen
                dc.l    0               ; Zeiger auf eigenen BitMap
                dc.w    100,50          ; Minimalwerte der Fenstergröße
                dc.w    200,100         ; Maximalwerte der Fenstergröße
                dc.w    1               ; Typ des Fensters
```

Programm 5.1: Öffnen und Schließen eines Windows

Wie man sieht, ist es gar nicht so schwer, ein eigenes Fenster zu öffnen. Man muß zunächst die benötigte Library öff-

nen, den Zeiger auf die NewWindow-Struktur übergeben und mittels der Basisadresse der Library die OpenWindow-Funktion anspringen. Damit es nicht sofort wieder geschlossen wird, muß eine Verzögerung eingebaut werden. Hierzu fragen wir solange die linke Maustaste (LMT) ab, bis diese gedrückt worden ist. Sie wird durch das sechste Bit der Adresse $BFE001 repräsentiert. Mit dem Befehl BTST (Bit TeST) können wir den Zustand der Taste abfragen. Wurde die LMT betätigt, so wird in den nächsten Zeilen erst das Fenster und dann die Library geschlossen.

Die NewWindow-Struktur haben wir bereits kennengelernt um ein Fenster zu öffnen. Doch die Window-Struktur, auf die uns Intuition einen Zeiger zurückgibt, haben wir bislang nicht beachtet. Da diese aber ziemlich wichtige und informative Einträge enthält, wollen wir sie schon an dieser Stelle erläutern.

Window-Struktur:

```
000   dc.l   *wd_NextWindow        ; Zeiger auf nächstes Window
004   dc.w    wd_LeftEdge          ; linke Ecke
006   dc.w    wd_TopEdge           ; obere Ecke
008   dc.w    wd_Width             ; Breite
010   dc.w    wd_Height            ; Höhe
012   dc.w    wd_MouseY            ; Y-Mauskoordinate
014   dc.w    wd_MouseX            ; X-Mauskoordinate
016   dc.w    wd_MinWidth          ; minimale Breite
018   dc.w    wd_MinHeight         ; minimale Höhe
020   dc.w    wd_MaxWidth          ; maximale Breite
022   dc.w    wd_MaxHeight         ; maximale Höhe
024   dc.l    wd_Flags             ; Window-Flags
028   dc.l   *wd_MenuStrip         ; Zeiger auf Menü-Struktur
032   dc.b   *wd_Title             ; Zeiger auf Titelzeile
036   dc.l   *wd_FirstRequester    ; Zeiger auf ersten Requester
040   dc.l   *wd_DMRequest         ; Zeiger auf Double-Menu-Req.
044   dc.w    wd_ReqCount          ; Zähler der Requester
046   dc.l   *wd_WScreen           ; Zeiger auf Screen
050   dc.l   *wd_RPort             ; Zeiger auf RastPort
054   dc.b    wd_BorderLeft        ;
055   dc.b    wd_BorderTop         ;
056   dc.b    wd_BorderRight       ;
057   dc.b    wd_BorderBottom      ;
058   dc.l   *wd_BorderRPort       ; Zeiger auf Border RastPort
062   dc.l   *wd_FirstGadget       ; Zeiger auf erstes Gadget
066   dc.l   *wd_Parent            ; vorhergehendes Fenster
070   dc.l   *wd_Descendant        ; nachfolgendes Fenster
074   dc.w   *wd_Pointer           ; Zeiger auf Mausdaten
078   dc.b    wd_PtrHeight         ; Höhe des Mauszeigers
079   dc.b    wd_PtrWidth          ; Breite des Mauszeigers
080   dc.b    wd_XOffset           ; X-Koordinate des HOTSPOT
081   dc.b    wd_YOffset           ; Y-Koordinate des HOTSPOT
082   dc.l    wd_IDCMPFlag         ; IDCMP-Flags
086   dc.l   *wd_UserPort          ; Zeiger auf UserPort
090   dc.l   *wd_WindowPort        ; Zeiger auf WindowPort
094   dc.l   *wd_MessageKey        ; Zeiger auf MessageKey
098   dc.b    wd_DetailPen         ; Farbe für Vordergrund
099   dc.b    wd_BlockPen          ; Farbe für Hintergrund
100   dc.l   *wd_CheckMark         ; Zeiger auf Grafikdaten
104   dc.b   *wd_ScreenTitle       ; Zeiger auf Screentitel
108   dc.w    wd_GZZMouseX         ; Mauskoordinaten für GZZ
110   dc.w    wd_GZZMouseY         ;
112   dc.w    wd_GZZWidth          ;
114   dc.w    wd_GZZHeight         ;
116   dc.b   *wd_ExtData           ; Zeiger auf externe Daten
120   dc.b   *wd_UserData          ; Zeiger auf eigene Daten
124   dc.l   *wd_WLayer            ; Zeiger auf Window-Layer
128   dc.l   *wd_IFont             ; Zeiger auf TextFontStruktur
132           wd_SIZEOF
```

***wd_NextWindow**
Zeiger auf das nächste Fenster, welches im Screen verwaltet wird.

wd_LeftEdge, wd_TopEdge, wd_Width, wd_Height
Position und Größe des Fensters. (Identisch mit NewWindow-Eintrag)

wd_MouseY, wd_MouseX
Mauskoordinaten bezogen auf die linke obere Ecke des Fensters.

wd_MinWidth, wd_MinHeight, wd_MaxWidth, wd_MaxHeight
Minimale und maximale Breite und Höhe des Fensters. (Identisch mit NewWindow-Eintrag)

wd_Flags
Flags für das Aussehen des Fensters (Identisch mit NewWindow-Eintrag).

***wd_MenuStrip**
Zeiger auf Menü-Struktur des Fensters (Siehe Abschnitt Menüs).

***wd_Title**
Zeiger auf Titel des Fensters (Identisch mit NewWindow-Eintrag).

***wd_FirstRequester**
Zeiger auf ersten Requester dieses Fensters (Siehe Abschnitt Requester).

***wd_DMRequest**
Zeiger auf eine Double-Menu-Request-Struktur (Siehe Abschnitt Requester).

wd_ReqCount
Anzahl der Requester, die schon geöffnet wurden.

***wd_WScreen**
Zeiger auf Struktur des Screens, zu dem das Fenster gehört.

***wd_RPort**
Zeiger auf Struktur des zum Fenster gehörigen RastPorts. Er ist nötig, um z.B. Grafiken und Text auszugeben.

wd_BorderLeft, wd_BorderTop, wd_BorderRight, wd_BorderBottom
Breitenangaben des Fensterrandes.

***wd_BorderRPort**
Zeiger auf RastPort-Struktur für Border (GZZ-Windows).

***wd_FirstGadget**
Zeiger auf erste Gadget-Struktur, die in das Fenster eingebunden werden soll (Identisch mit NewWindow-Eintrag).

***wd_Parent, *wd_Descendant**
Zeiger auf die Struktur des vorhergehenden bzw. nachfolgenden Fensters.

***wd_Pointer**
Adresse der Grafikdaten für den Mauszeiger. Jedes Fenster kann die Form des Mauszeigers speziell definieren.

wd_PtrHeight, wd_PtrWidth
Breite und Höhe des Maus-Sprites. Dabei ist zu beachten, daß die Sprites eine limitierte Breite haben (1 Word = 16 Bit (Punkte)).

wd_XOffset, wd_YOffset
Position des HOTSPOT relativ zur oberen linken Ecke der Grafikdaten. Als HOTSPOT bezeichnet man einen Punkt der Mauszeigerdaten, der als auslösender Punkt angesehen wird.

wd_IDCMPFlag
Intuition Direct Communications Message Ports - Flags. (Identisch mit NewWindow-Eintrag).

***wd_UserPort**
Zeiger auf MessagePort-Struktur für Datenaustausch.

***wd_WindowPort**
Zeiger auf MessagePort-Struktur für Datenaustausch.

***wd_MessageKey**
Zeiger auf MessagePort-Struktur für Datenaustausch.

wd_DetailPen, wd_BlockPen
Farbtabellennummer des Vorder- (DetailPen) und Hintergrund (BlockPen). (Identisch mit NewWindow-Einträgen).

***wd_CheckMark**
Zeiger auf Grafikdaten für den Menü-Haken. Der Menü-Haken wird benötigt, um einen Menüpunkt als ausgewählt zu kennzeichnen (Identisch mit NewWindow-Eintrag).

***wd_ScreenTitle**
Zeiger auf Zeichenkette des Screentitels.

wd_GZZMouseX, wd_GZZMouseY
Koordinatenangabe des Mauszeigers im Fenster abzüglich des bei einem GZZ-Window eingebundenen Randes.

wd_GZZWidth, wd_GZZHeight
Breiten- und Höhenangabe bei GIMMEZEROZERO-Windows.

***wd_ExtData**
Zeiger auf externe Daten.

***wd_UserData**
Zeiger auf Daten, die vom Benutzer definiert, benutzt und eingebunden werden können (Identisch mit NewWindow-Eintrag).

***wd_WLayer**
Addresse der Layer-Struktur, die für das Fenster verantwortlich ist.

***wd_IFont**
Zeiger auf die TextFont-Struktur des Fonts, der für die Ausgabe in das Fenster benutzt werden soll.

Sicherlich sind die vielen Einträge an dieser Stelle sehr verwirrend. Doch werden sie nach und nach in diesem Kapitel erklärt.

5.1.1 Nachrichten empfangen

Das letzte Programm ist im Bezug auf die Mausabfrage sehr einfach. Nun wollen wir etwas tiefer einsteigen und die Meldungen, die unser Programm empfängt, auswerten. Hier kommen auch wieder die IDCMP-Flags ins Gespräch. Sie geben an, welche Ereignisse, die unser Fenster betreffen, von Intuition an uns weitergeleitet werden sollen.

Man muß sich das wie folgt vorstellen: Nachdem wir unser Fenster erfolgreich geöffnet haben, bewegt der Benutzer den Mauszeiger auf das Close-Gadget des Windows und drückt die linke Maustaste.

Das hat das System natürlich mitbekommen und schickt eine Nachricht an den MessagePort, eine Art Briefkasten (wird gleich noch erklärt), des betreffenden Fensters. In dieser Nachricht (IntuiMessage-Struktur) sind einige wichtige Informationen enthalten, so z.B. den Eintrag namens "Class", in dem das IDCMP-Flag gesetzt worden ist, welches die Meldung ausgelöst hat.

Von unserem Programm aus gesehen, spielt sich die Sache etwas anders ab. Wenn wir das Fenster geöffnet haben, brauchen wir nur noch auf eine Meldung von Intuition zu warten. Haben wir über den MessagePort unseres Fensters eine Meldung erhalten, können wir nun die IntuiMessage-Struktur auswerten. Danach müssen wir die Nachricht noch bestätigen, damit Intuition weiß, das alles glatt gelaufen ist.

Die besagte IntuiMessage-Struktur, dessen Einträge man kennen sollte, hat folgendes Aussehen:

IntuiMessage-Struktur:

```
00    dc.l   *ln_Succ          ;┐         ┐
04    dc.l   *ln_Pred          ;│         │
08    dc.b    ln_Type          ;│ Node    │
09    dc.b    ln_Pri           ;│         │
10    dc.l   *ln_Name          ;┘         │ Message-Struktur
                                          │
14    dc.l   *mn_ReplyPort     ;          │
18    dc.w    mn_Length        ;          ┘

20    dc.l    im_Class         ; IDCMP-Flag der Nachricht
```

```
24      dc.w     im_Code            ; Nachrichten-abhängige Daten
26      dc.w     im_Qualifier       ;
28      dc.l     im_IAddress        ; Zeiger auf den Auslöser
32      dc.w     im_MouseX          ; Mauskoordinate (X)
34      dc.w     im_MouseY          ; Mauskoordinate (Y)
36      dc.l     im_Seconds         ; Sekunden
40      dc.l     im_Micros          ; Mikros
44      dc.l    *im_IDCMPWindow     ; Zeiger auf Fenster
48      dc.l    *im_SpecialLink     ; Systemspezifisch
52               im_SIZEOF
```

ln_Succ bis mn_Length
Diese ersten sieben Einträge sind für das System relevant. Auf ihre Bedeutung wird im Kapitel Exec näher eingegangen.

im_Class
Wie schon erwähnt enthält der Eintrag im_Class das IDCMP-Flag, welches das auslösende Ereignis angibt.

im_Code
Der in Code übergebene Wert ist von der ausgelösten Nachricht abhängig. Die Bedeutung wird deshalb erst nachher besprochen.

im_Qualifier
Tastencode einer gedrückten Qualifier-Taste.

im_IAddress
An dieser Stelle kann z.B. ein Zeiger stehen, der auf das ausgewählte Gadget verweist. Dadurch kann man es an Hand seiner ID-Nummer identifizieren.

im_MouseX, im_MouseY
Koordinaten des Mauszeigers zur Zeit, als die Meldung ausgelöst wurde.

im_Seconds, im_Micros
Zeitpunkt, an dem die Nachricht ausgelöst wurde. Die Werte beziehen sich auf die System-Uhr.

***im_IDCMPWindow**
Zeiger auf das Fenster, auf das sich diese Meldung bezieht.

***im_SpecialLink**
Der Eintrag SpecialLink ist nur für das System relevant.

Um nun eine Nachricht von unserem MessagePort zu holen, müssen wir uns eine Funktion der Exec-Library "ausleihen". Sie heißt GetMsg und gibt uns nach ihrem Aufruf die Adresse einer Nachrichten-Struktur zurück oder eine Null.

GetMsg	=	-372 (Exec-Library)

*Port   a0 < Zeiger auf einen MessagePort.

Message   d0 > Zeiger auf eine Message-Struktur oder, wenn keine weitere Nachricht anliegt, eine Null.

Erklärung   Mit Hilfe der Funktion GetMsg wird die nächste Meldung, die der angegebene MessagePort enthält, aus der Liste der Nachrichten des Ports entfernt und der Zeiger auf diese Struktur in d0 zurückgegeben.

Wie man sieht, benötigt die Funktion nur einen Parameter, einen Zeiger auf einen MessagePort. Glücklicherweise hat Intuition beim Öffnen des Fensters eine solche Struktur schon für uns angelegt. Also brauchen wir uns darum nicht mehr zu kümmern. Den Zeiger können wir, wie der folgende Programmausschnitt zeigt, aus der Window-Struktur unseres Fensters leicht auslesen (er ist im Eintrag UserPort enthalten).

```
        ...
        move.l   WindowHD,a0      ; Zeiger auf Window nach a0
        move.l   86(a0),UPort     ; UserPort Adresse auslesen
        ...

UPort:        dc.l    0                ; Platz für den UserPort-Zeiger
```

Bild 5.1: Ermitteln der UserPort-Adresse eines Windows

Sollten wir mit GetMsg eine Nachricht empfangen haben, speichern wir den Zeiger zunächst in einer Variablen. Wenn kein Ereignis stattgefunden hat, erhalten wir keinen Zeiger, sondern eine Null zurück. In diesem Fall wird nicht zu der Routine verzweigt, die für die Untersuchung der IntuiMessage-Struktur verantwortlich ist.

Wir könnten jetzt immer wieder mit der GetMsg-Funktion kontrollieren, ob eine Nachricht angekommen ist oder nicht. Diese Möglichkeit ist jedoch nicht besonders professionell und braucht zudem auch viel Rechenzeit. Eine bessere Lösung bietet eine weitere Exec-Funktion mit dem Namen WaitPort.

WaitPort	=	**-384 (Exec-Library)**

***Port** **a0** < Zeiger auf eine Port-Struktur.

Message **d0** > Zeiger auf die Message-Struktur, die empfangen worden ist.

Erklärung Durch die Funktion WaitPort wird das Programm solange unterbrochen, bis es eine Nachricht von Intuition geschickt bekommt. Die Adresse der IntuiMessage-Struktur erhält man zwar zurück, doch muß man sie durch die GetMsg-Funktion aus der Liste nehmen.

WaitPort schickt das Programm solange "schlafen", bis Intuition eine Nachricht an den angegebenen MessagePort sendet. Dann erst wird das Programm fortgesetzt. Auf den genauen Ablauf der WaitPort-Funktion gehen wir im Exec-Kapitel näher ein.

Falls ein Ereignis stattgefunden hat, erhalten wir einen Zeiger auf eine Nachrichten-Struktur in d0. Da aber WaitPort die Nachricht nicht aus der Liste der Nachrichten auskoppelt, müssen wir dazu die GetMsg-Funktion verwenden, damit die Nachricht nicht zweimal behandelt wird.

Der vollständige Teil für die Abfrage des Ports sieht dann wie folgt aus.

```
GetMsg          =       -372
WaitPort        =       -384


        ...
MessageLoop:
        move.l  ExecBase,a6

        move.l  UPort,a0        ; nächste Meldung des UPorts
        jsr     GetMsg(a6)      ; abholen.

        move.l  d0,Message      ; Zeiger zwischenspeichern
        bne     MessageBranch   ; Wurde wirklich ein Wert <>0
                                ; gespeichert ?

        move.l  UPort,a0        ; Nein, dann warten bis
        jsr     WaitPort(a6)    ; unser UserPort eine
        bra     MessageLoop     ; Nachricht erhält
        ...

Message:        dc.l    0               ; Platz für den Zeiger auf eine
                                        ; IntuiMessage-Struktur
```

Bild 5.2: Abfrage eines Window-MessagePorts

Nachdem wir besprochen haben, wie eine Nachricht empfangen wird, untersuchen wir jetzt deren Auswertung. Dazu haben wir folgende Unterroutine angelegt.

```
ReplyMsg      =       -378

        ...

MessageBranch:
        move.l   Message,a0       ; Zeiger auf IntuiMessage
        move.l   20(a0),Class     ; IDCMP-Kennung auslesen

        move.l   Message,a1       ; Meldung bestätigen
        jsr      ReplyMsg(a6)

        cmp.l    #$200,Class      ; Close-Gadget betätigt ?
        beq      Exit             ; Ja dann Programm beenden

        bra      MessageLoop      ; Neuer Versuch

        ...

Class:        dc.l    0               ; Variable für das IDCMP-Flag der
                                      ; Nachricht
```

Bild 5.3: Auswertung einer Message

Zunächst lesen wir aus der IntuiMessage-Struktur den Eintrag im_Class aus, der das IDCMP-Flag enthält, welches die Nachricht ausgelöst hat, und speichern es in der Variablen Class. Danach müssen wir die Nachricht bestätigen. Dazu schicken wir die IntuiMessage-Struktur an den angegebenen ReplyPort zurück. Hierzu gibt es natürlich auch eine Funktion in der der Exec-Library.

ReplyMsg	=	**-378 (Exec-Library)**

***Message**	**a1**	<	Zeiger auf Message-Struktur, die bestätigt werden soll.
Erklärung			ReplyMsg sendet die angegebene Nachrichten-Struktur an den ReplyPort, der in der Message-Struktur eingetragen ist, zurück.

Nachdem wir die Formalitäten erledigt haben, können wir nun den Classwert untersuchen und entsprechend dem Ereignis reagieren. Im Programm warten wir z.B. darauf, daß das Close-Gadget des Fensters aktiviert worden ist. Erst dann wird das Fenster und die Library geschlossen.

Natürlich können auch andere IDCMP-Flags abgefragt werden. Doch sollte man nie vergessen, die "Nachrichten-Sperre" durch das Setzen des zugehörigen Flags im IDCMPFlags-Eintrag der NewWindow-Struktur aufzuheben. Sonst wartet man vergeblich auf eine Nachricht!

Jetzt erwarten Sie sicher ein Demonstrationsprogramm. Aus Platzgründen haben wir es jedoch auf die Diskette verbannt. Außerdem sind die folgenden Programme nach dem gleichen Schema aufgebaut.

Zum Schluß listen wir jetzt noch alle wichtigen Funktionen zur Verwaltung der Fenster auf, die wir bis jetzt noch nicht besprochen haben. Einige dieser Funktionen sind in dem Demonstrations-Programm zu diesem Abschnitt aufgeführt.

ActivateWindow	=	**-450 (Intuition-Library)**

***Window** **a0** < Zeiger auf das Window, das aktiviert werden soll.

Erklärung Das angegebene Fenster wird aktiviert. Dies kann auch schon durch das ACTIVATE-Flag beim Öffnen des Fensters erreicht werden.

BeginRefresh	=	**-354 (Intuition-Library)**

***Window** **a0** < Zeiger auf eine Window-Struktur.

Erklärung Meldet Intuition, daß der Fensterinhalt vom Programm erneuert wird. Sinnvoll ist diese Funktion in bezug auf das SIMPLE_REFRESH Flag, bei dem die Regenerierung des Fensters selbständig vom Programm übernommen werden muß (siehe auch EndRefresh).

ClearPointer	=	**-30 (Intuition-Library)**

***Window** **a0** < Zeiger auf eine Window-Struktur

Erklärung Mauszeiger des betreffenden Windows wird gelöscht (siehe auch SetPointer).

EndRefresh = -366 (Intuition-Library)

*Window	a0	<	Zeiger auf eine Window-Struktur
Complete	d0	<	Nachdem die Funktion aufgerufen worden ist, enthält die Variable, deren Adresse in d0 übergeben wurde, eine Null, wenn das Fenster nicht vollständig erneuert wurde.

Erklärung Teilt Intuition mit, daß der Refreshstatus beendet ist (siehe auch BeginRefresh).

ModifyIDCMP = -150 (Intuition-Library)

*Window	a0	<	Zeiger auf eine Window-Struktur
Flags	d0	<	Neue IDCMPFlags, die eingesetzt werden sollen.

Erklärung Mit Hilfe dieser Funktion ist es möglich, die IDCMP-Flags des Fensters zu verändern (siehe auch ReportMouse).

MoveWindow = -168 (Intuition-Library)

*Window	a0	<	Zeiger auf eine Window-Struktur
dx	d0	<	X-Wert, um den das Fenster verschoben werden soll. Es sind auch negative Werte erlaubt.
dy	d1	<	Y-Wert, um den das Fenster verschoben werden soll. Es sind auch negative Werte erlaubt.

Erklärung Das ausgewählte Fenster wird um die angegebene Strecke in X- und in Y-Richtung veschoben.

RefreshWindowFrame = -456 (Intuition-Library)

*Window	a0	<	Zeiger auf eine Window-Struktur.

Erklärung Der Rahmen des Fensters wird neu gezeichnet.

ReportMouse = -234 (Intuition-Library)

***Window**	**a0**	<	Zeiger auf eine Window-Struktur
Boolean	**d0**	<	Null, wenn das REPORTMOUSE-Flag gelöscht werden soll; sonst wird es gesetzt.

Erklärung

Schaltet das IDCMP-Flag REPORTMOUSE nachträglich ein oder aus. Dieses Flag bewirkt, daß die Mausposition dem Programm laufend mitgeteilt wird (siehe auch ModifyIDCMP).

SetPointer = -270 (Intuition-Library)

***Window**	**a0**	<	Zeiger auf eine Window-Struktur.
***Pointer**	**a1**	<	Zeiger auf die Sprite-Daten für den Mauszeiger.
Height	**d0**	<	Höhe des Mauszeigers.
Width	**d1**	<	Breite des Mauszeigers.
HotX	**d2**	<	X-Koordinate des HOTSPOT.
HotY	**d3**	<	Y-Koordinate des HOTSPOT.

Erklärung

SetPointer setzt einen nur für dieses Window zuständigen Mauszeiger (siehe auch ClearPointer).

SetWindowTitles = -276 (Intuition-Library)

***Window**	**a0**	<	Zeiger auf eine Window-Struktur
***WinTitle**	**a1**	<	Zeiger auf eine Zeichenkette für den Fenstertitel
***ScrTitle**	**a2**	<	Zeiger auf eine Zeichenkette für den Screentitel

Erklärung

Mit SetWindowTitles kann man den Titel des Fensters sowie den Titel des Screens, der abhängig vom Fenster angezeigt wird, setzen. Wird anstelle des Zeigers auf die Zeichenkette eine 0 eingetragen, so wird der Titel gelöscht. Der Wert -1 bewirkt, daß der alte Titel beibehalten wird.

SizeWindow = -288 (Intuition-Library)

***Window**	**a0**	<	Zeiger auf eine Window-Struktur
dx	**d0**	<	Relativer Wert, bezogen auf die derzeitige Fenstergröße, um die die Breite

			verändert werden soll. Es sind auch negative Werte erlaubt.
dy	d1	<	Relativer Wert, bezogen auf die derzeitige Fenstergröße, um die die Höhe verändert werden soll. Es sind auch negative Werte erlaubt.

Erklärung

Die Größe des Fensters kann durch diese Funktion neu eingestellt werden. Dazu müssen zwei Werte angegeben werden, welche die Veränderung relativ zur derzeitigen Größe darstellen (siehe auch WindowLimits).

ViewPortAddress = -300 (Intuition-Library)

*Window	a0	<	Zeiger auf eine Window-Struktur
ViewPort	d0	>	Adresse der ViewPort-Struktur des Fensters.

Erklärung

Durch die Funktion ViewPortAddress wird die Adresse der ViewPort-Struktur ermittelt, die für das Fenster verantwortlich ist.

WindowLimits = -318 (Intuition-Library)

*Window	a0	<	Zeiger auf eine Window-Struktur
WMin	d0	<	Wert für die minimale Breite
HMin	d1	<	Wert für die minimale Höhe
WMax	d2	<	Wert für die maximale Breite
HMax	d3	<	Wert für die maximale Höhe
Success	d0	>	Wenn die derzeitige Größe außerhalb der Limitierung liegt, erhält man, nachdem man die Funktion aufgerufen hat, eine Null zurück.

Erklärung

Die Limitierung der Ausdehnung des Fensters wird neu gesetzt. Der zurückgegebene Wert informiert, ob sich die Größe des Fenster in den angegebenen Grenzen befindet (siehe auch SizeWindow).

WindowToBack = -306 (Intuition-Library)

*Window	a0	<	Zeiger auf eine Window-Struktur.

Erklärung Das angegebene Fenster wird in den Hintergrund gesetzt (siehe auch WindowToFront).

WindowToFront	=	**-312 (Intuition-Library)**

***Window** **a0** < Zeiger auf eine Window-Struktur

Erklärung Das angegebene Fenster wird in den Vordergrund geholt (siehe auch WindowToBack).

5.2 Screens

Nun zu den Screens. Sie sind eine Art virtueller Bildschirm, auf dem beliebig viele Windows geöffnet werden können. Die Screens, die scheinbar aufeinander liegen, können jede Grafikauflösung, die der Amiga zur Verfügung stellt, annehmen. Die Anzahl ist dabei lediglich durch die Größe des Chip-Ram-Speichers begrenzt.

Wie schon bei den Windows, gibt es auch bei den Screens eine Struktur, worin die Definitionen eingetragen werden müssen. Sinngemäß heißt sie hier NewScreen-Struktur.

NewScreen-Struktur:

```
00    dc.w    ns_LeftEdge         ; X-Koordinate des Screens
02    dc.w    ns_TopEdge          ; Y-Koordinate des Screens
04    dc.w    ns_Width            ; Breite
06    dc.w    ns_Height           ; Höhe
08    dc.w    ns_Depth            ; Anzahl BitPlanes
10    dc.b    ns_DetailPen        ; Farben für den Vordergrund
11    dc.b    ns_BlockPen         ; und den Hintergrund
12    dc.w    ns_ViewModes        ; Auflösung
14    dc.w    ns_Type             ; Screen-Typ
16    dc.l   *ns_Font             ; TextAttr-Struktur
20    dc.l   *ns_DefaultTitle     ; Zeiger auf Titelzeile
24    dc.l   *ns_Gadgets          ; Custom-Gadgets
28    dc.l   *ns_CustomBitMap     ; eigene BitMap-Struktur
32            ns_SIZEOF
```

ns_LeftEdge, ns_TopEdge, ns_Width, ns_Height
Position und Größe des Screens.

ns_Depth
Anzahl der BitPlanes, die für den Screen verwendet werden sollen. Dadurch ist auch die Anzahl der Farben festgelegt.

$$(\text{Anzahl Farben}) = 2^{(\text{Anzahl BitPlanes})}$$

Bei zwei BitPlanes, kann man z.B. 2^2 = 4 Farben verwenden.

ns_DetailPen, ns_BlockPen
Farbtabellennummer der Vorder- und Hintergrundfarbe. Die Farben, die zur Verfügung stehen, werden in einer Farbpalette abgelegt, aus der man eine Farbe auswählen kann. Die Einstellung ist abhängig von der in Depth angegebenen Tiefe. So können wir, bei vier Farben, maximal die Farbe 3 (es wird von 0 bis 3 gezählt) einstellen.

ns_ViewModes
Auflösung des Screens. Es gibt folgende Möglichkeiten:

View-Mode	Wert	Bedeutung
GENLOCK_VIDEO	$0002	Bindet eine externe Signalquelle ein
EXTRA_HALFBRITE	$0080	64-Farben Modus
DUALPF	$0400	Dual-Playfield
Hold-And-Modify	$0800	HAM-Modus (4096 Farben)
VP_HIDE	$2000	Kein Bild
SPRITES	$4000	Screen mit Hardware-Sprites
HIRES	$0004	Verdoppelt Auflösung in X-Richtung
LACE	$8000	Verdoppelt Auflösung in Y-Richtung

GENLOCK_VIDEO Es wird das Videosignal eines Genlock Video Interfaces in den Hintergrund eingebunden.

EXTRA_HALFBRITE Dieser Modus erlaubt die Verarbeitung von 6 BitMaps, also 64 Farben. Dabei sind nur die ersten 32 Farben frei wählbar, die restlichen 32 entsprechen den ersten mit halber Helligkeit.

DUALPF Dual-Playfield

Hold-And-Modify Dieser Modus, auch HAM-Modus genannt, erlaubt die Darstellung von 4096 Farben gleichzeitig! Hierzu muß ein Trick angewendet werden, da die maximale Anzahl der Bitplanes nur für 2^6 = 64 Farben reichen würde. Der Trick besteht darin, daß die Kombination aus den Bits der 5. und 6. Plane die Verwendung der 1. bis 4. festlegt.

Kombination der 5. und 6. BP	Verhalten der Planes 1 bis 4
00	(normal) Wahl des Farbregisters
01	Rotwert des letzten gewählten Registers
10	Grünwert des letzten gewählten Registers
11	Blauwert des letzten gewählten Registers

Für diesen Modus müssen mindestens 5 BitPlanes benutzt werden.

VP_HIDE Screen wird nicht dargestellt.

SPRITES Die Benutzung von Sprites wird erlaubt.

HIRES	Die Auflösung in X-Richtung wird von 320 auf 640 Punkte verdoppelt.
LACE	Die Auflösung in Y-Richtung wird von 256 auf 512 Punkte verdoppelt. Hierbei wird das Bild nicht in einem Zuge aufgebaut, sondern erst die geraden Zeilen und dann die ungeraden. Dadurch verdoppelt sich die Zeit des Bildaufbaus und es entsteht bei Monitoren mit einer zu geringen Wiederholfrequenz (z.B. 50Hz) das berüchtigte Flackern des Bildes.

ns_Type
Typ des Screens, der erstellt werden soll.

Screentyp	Wert	Bedeutung
WBENCHSCREEN	$0001	Screen ist der WBench-Screen
CUSTOMSCREEN	$000F	Screen ist Custom-Screen
CUSTOMBITMAP	$0040	Keine BitMap erstellen
SCREENBEHIND	$0080	Screen im Hintergurnd öffnen
SCREENQUIET	$0100	System-Gadgets/Titelzeile abschalten
SHOWTITLE	$0010	Intuition-intern
BEEPING	$0020	Intuition-intern

WORKBENCHSCREEN	Screen ist die WorkBench (nur von Intuition benutzt)
CUSTOMSCREEN	Es wird ein eigener Screen (CUSTOMSCREEN) erstellt.
CUSTOMBITMAP	Intuition soll keine BitMap für unseren Screen anlegen. Das kann wichtig sein, wenn man eine eigene BitMap einbinden will.
SCREENBEHIND	Normalerweise wird ein neuer Screen als erster, also als sichtbarer Screen erstellt. Dies kann man durch die Einstellung SCREENBEHIND verhindern. Dann wird der Screen nicht im Vordergrund, sonder im Hintergrund geöffnet.
SCREENQUIET	SCREENQUIET unterdrückt das Zeichnen der System-Gadgets und der Titelzeile des Screens. Wenn jedoch die rechte Maustaste gedrückt wird, erscheint das Menü in der Titelzeile, wird aber nicht wieder gelöscht. Um diesen unschönen Effekt zu verhindern, kann man z.B. durch die Einstellung RMBTRAP im Window-Flag-Eintrag der NewWindow-Struktur das Anwählen der Menüzeile verhindern.
SHOWTITLE	Die letzten beiden Flags SHOWTITLE und BEEPING werden von Intuition gesetzt
BEEPING	(Screen blinkt)

ns_TextAttr
Zeiger auf eine TextAttr-Struktur, durch die der Zeichensatz, mit dem auf den Screen geschrieben werden soll, bestimmt wird. Soll der Standardzeichensatz verwendet werden, muß der Zeiger mit Null initialisiert werden.

ns_DefaultTitle
Adresse einer Zeichenkette, die den DafaultTitle angibt. Setzt man eine Null ein, wird kein Titel benutzt.

ns_Gadgets
ACHTUNG: Dieser Eintrag wird (noch) nicht von Intuition unterstützt. Hier sollte immer eine Null eingetragen werden, da man nie weiß, ob nicht in der nächsten Kickstart-Version diese Funktion berücksichtigt ist.

ns_CustomBitMap
Zeiger auf eine eigene initialisierte BitMap-Struktur, die an Stelle der, von Intuition angelegten, verwendet werden soll (siehe auch Eintrag Type (CUSTOMBITMAP)). Wenn man auf eine eigene BitMap-Struktur verzichtet, so setzt man den Eintrag auf Null und Intuition richtet selbständig eine BitMap-Struktur ein.

Nachdem wir die NewScreen-Struktur besprochen haben, kommen wir zu den Funktionen zum Öffnen eines Screens.

OpenScreen	**=**	**-198 (Intuition-Library)**

***NewScreen**	**a0**	<	Zeiger auf NewScreen-Struktur
Screen	**d0**	>	Zeiger auf Screen-Struktur
Erklärung			Mit Hilfe der OpenScreen-Funktion kann ein vordefinierter Screen geöffnet werden. Außerdem legt Intuition, wie schon bei den Windows, eine spezielle Screen-Struktur an und gibt den Zeiger auf sie zurück.

Wie man sieht, erhalten wir einen Zeiger auf eine von Intuition angelegte Struktur. Dieser Zeiger ist sehr wichtig, da alle Screen-Funktionen auf ihn zurückgreifen. So auch die CloseScreen-Funktion, welche das Schließen des Screens für uns erledigt.

CloseScreen	**=**	**-72 (Intuition-Library)**

***Screen**	**a0**	<	Zeiger auf Screen-Struktur
Erklärung			Die Funktion CloseScreen schließt den angegebenen Screen.

Jetzt, nachdem wir alle wichtigen Funktionen kennengelernt haben, wollen wir diese praktisch anwenden und einen Screen und ein Window öffnen.

```
* Programm 5.4: Screen und Window öffnen

ExecBase      =       4
OldOpenLib    =       -408
CloseLib      =       -414
GetMsg        =       -372
ReplyMsg      =       -378
WaitPort      =       -384
OpenScreen    =       -198
CloseScreen   =       -66
OpenWindow    =       -204
CloseWindow   =       -72

Start:
        move.l  ExecBase,a6      ; Intuition-Lib öffnen
        lea     IntName,a1
        jsr     OldOpenLib(a6)
        move.l  d0,IntBase
        beq     IntError

        move.l  IntBase,a6       ; Screen öffnen
        lea     ScreenArgs,a0    ; NewScreen-Struktur
        jsr     OpenScreen(a6)
        move.l  d0,ScreenHD      ; Zeiger auf Screen in
        beq     ScrError         ; die NewWindow-Struktur
                                 ; eintragen
        lea     WindowArgs,a0
        jsr     OpenWindow(a6)   ; Fenster öffnen
        move.l  d0,WindowHD
        beq     WinError

        move.l  WindowHD,a0      ; UserPort aus der
        move.l  86(a0),UPort     ; Window-Struktur lesen

MessageLoop:
        move.l  ExecBase,a6

        move.l  UPort,a0         ; Meldung angekommen ?
        jsr     GetMsg(a6)
        move.l  d0,Message
        bne     MessageBranch    ; Ja, dann verarbeiten

        move.l  UPort,a0         ; Nein, dann warten
        jsr     WaitPort(a6)
        bra     MessageLoop

Exit:
        move.l  IntBase,a6       ; Fentser schließen
        move.l  WindowHD,a0
        jsr     CloseWindow(a6)
```

```
WinError:
        move.l   ScreenHD,a0    ; Screen schließen
        jsr      CloseScreen(a6)

ScrError:
        move.l   ExecBase,a6    ; Library schließen
        move.l   IntBase,a1
        jsr      CloseLib(a6)

IntError:
        rts                     ; Ende !

MessageBranch:
        move.l   Message,a0     ; Class Eintrag der
        move.l   20(a0),Class   ; IntuiMessage-Struktur
                                ; speichern

        move.l   Message,a1     ; Nachricht beantworten
        jsr      ReplyMsg(a6)

        cmp.l    #$200,Class    ; CloseWindow betätigt ?
        beq      Exit           ; Ja, dann Ende

        bra      MessageLoop    ; nächste Nachricht holen

* Datenbereich

Class:         dc.l    0             ; IDCMP-Flag der Nachricht
Message:       dc.l    0             ; Zeiger auf die IntuiMessage
UPort:         dc.l    0             ; Zeiger auf den UserPortt
IntBase:       dc.l    0             ; Basisadresse der IntuitionLib
WindowHD:      dc.l    0             ; Zeiger auf das Fenster

IntName:       dc.b    "intuition.library",0
               even

WinName:       dc.b    "Close-Gadget beendet das Programm !",0
               even

ScrName:       dc.b    "CUSTOMSCREEN",0
               even

WindowArgs:
               dc.w    90,10               ; Definitionen des Fensters
               dc.w    460,200
               dc.b    1,3
               dc.l    $200                ; IDCMP-Flag
               dc.l    $1100F
               dc.l    0
               dc.l    0
               dc.l    WinName       ; Zeiger auf Fensternamen
ScreenHD:      dc.l    0             ; <= Hier wird der Zeiger auf
               dc.l    0             ;    den Screen eingetragen.
               dc.w    100,50
               dc.w    200,100
               dc.w    15            ; <= CUSTOMSCREEN !!
```

```
ScreenArgs:                             ; Definitionen des Screens
              dc.w    0,0               ; X/Y
              dc.w    640,256           ; Breite Höhe
              dc.w    2                 ; Tiefe
              dc.b    2,1               ; Farben
              dc.w    $8000             ; Modus
              dc.w    15                ; Screen-Typ
              dc.l    0                 ; Zeichensatz
              dc.l    ScrName           ; Titelzeile
              dc.l    0                 ; Gadgets
              dc.l    0                 ; eigene BitMap
```

Programm 5.4: Screen und Window öffnen

Das Listing dürfte eigentlich keine Schwierigkeiten machen, da wir die Nachrichtenbehandlung schon besprochen haben. Nur eine Passage könnte ihnen unbekannt vorkommen. Da ja das Fenster, welches wir öffnen wollen, auf unserem Screen erscheinen soll, müssen wir dies Intuition auch mitteilen. Dazu sind zwei Einträge in der NewWindow-Struktur zu ändern. Zunächst muß der letzte Eintrag (Type) der Window-Struktur von 1 (= WBENCHSCREEN) auf 15 (= CUSTOMSCREEN) geändert werden. Als zweites müssen wir den Zeiger auf die Screen-Struktur in die NewWindow-Struktur an die Stelle "*nw_Screen" eintragen. Da Intuition logischerweise diese Struktur erst anlegt, wenn wir den Screen öffnen wollen, müssen wir diesen Eintrag während des Ablaufes initialisieren.

Wie gesagt erstellt Intuititon eine eigene Struktur, die zur Verwaltung des Screens benutzt wird. Unsere Bauanleitung (NewScreen-Struktur) wird dann nicht mehr benötigt. Der Vollständigkeit wegen folgt nun die Screen-Struktur.

Screen-Struktur:

```
000   dc.l   *sc_NextScreen       ; Zeiger auf nächsten Screen
004   dc.l   *sc_FirstWindow      ; Zeiger auf erstes Window
008   dc.w    sc_LeftEdge         ; linke Ecke
010   dc.w    sc_TopEdge          ; rechte Ecke
012   dc.w    sc_Width            ; Breite
014   dc.w    sc_Height           ; Höhe
016   dc.w    sc_MouseY           ; Y-Mauskoordinaten
018   dc.w    sc_MouseX           ; X-Mauskoordinaten
020   dc.w    sc_Flags            ; Screen-Flags
022   dc.l   *sc_Title            ; Zeiger auf Titelstring
026   dc.l   *sc_DefaultTitle     ; Default-Titelstring
030   dc.b    sc_BarHeight        ; Höhe der Titelleiste
031   dc.b    sc_BarVBorder       ; vertikaler Rand
032   dc.b    sc_BarHBorder       ; horizontaler Rand
033   dc.b    sc_MenuVBorder      ; vertikaler Rand (Menü)
```

```
034   dc.b    sc_MenuHBorder      ; horizontaler Rand (Menü)
035   dc.b    sc_WBorTop          ; Breite, Window-Rand oben
036   dc.b    sc_WBorLeft         ; Breite, Window-Rand links
037   dc.b    sc_WBorRight        ; Breite, Window-Rand rechts
038   dc.b    sc_WBorBottom       ; Breite, Window-Rand unten
039   dc.b    sc_KludgeFill00     ; Füllbyte
040   dc.l   *sc_Font             ; TextAttr-Struktur
044   ds.b    sc_ViewPort,40      ; ViewPort-Struktur
084   ds.b    sc_RastPort,100     ; RastPort-Struktur
184   ds.b    sc_BitMap,40        ; BitMap-Struktur
224   ds.b    sc_LayerInfo,92     ; LayerInfo-Struktur
316   dc.l   *sc_FirstGadget      ; Zeiger auf erstes Gadget
320   dc.b    sc_DetailPen        ; Farbtab.nr. für Vordergrund
321   dc.b    sc_BlockPen         ; Farbtab.nr. für Hintergrund
322   dc.w    sc_SaveColor0       ; BEEPING SaveMem
324   dc.l   *sc_BarLayer         ; Zeiger auf Layer
328   dc.l   *sc_ExtData          ; Zeiger auf externe Daten
332   dc.l   *sc_UserData         ; Zeiger für UserDaten
336           sc_SIZEOF
```

***sc_NextScreen**
Zeiger auf den nächsten Screen, der dargestellt werden soll.

***sc_FirstWindow**
Zeiger auf das erste Fenster, welches im Screen dargestellt wird.

sc_LeftEdge, sc_TopEdge, sc_Width, sc_Height
Position und Größe des Screens (übernommen aus NewScreen-Struktur).

sc_MouseY, sc_MouseX
Mauskoordinaten relativ zur oberen linken Ecke des Screens.

sc_Flags
Angabe über das Aussehen des Screens (übernommen aus New-Screen-Struktur).

***sc_Title**
Zeiger auf Titelzeile für den Screen, die von einem Fenster festgelegt worden ist (übernommen aus NewWindow-Struktur).

***sc_DefaultTitle**
Zeiger auf Titelzeile, die in der NewScreen-Struktur festgelegt worden ist.

sc_BarHeight
Höhe der Titelleiste.

sc_BarVBorder, sc_BarHBorder
Breite des veritkalen und horizontalen Randes.

sc_MenuVBorder, sc_MenuHBorder
Horizontale und vertikale Randbreite der Menüs.

sc_WBotTop, sc_WBorLeft, sc_WBorRight
Breite des oberen, des linken und des rechten Fensterrandes.

sc_KludgeFill00
Der Eintrag sc_KludgeFill00 dient lediglich als Füllbyte, um den PC wieder auf Wortgrenze zu bringen.

***sc_Font**
Zeiger auf eine TextAttr-Struktur, die den Zeichensatz für den Screen angibt.

sc_ViewPort
An dieser Stelle ist eine komplette ViewPort-Struktur eingebunden, die Daten wie z.B. die Farbtabelle enthält.

sc_RastPort
Komplette RastPort-Struktur des Screens.

sc_BitMap
Komplette BitMap-Struktur des Screens.

sc_LayerInfo
Komplette Layer_Info-Struktur des Screens.

***sc_FirstGadget**
Zeiger auf erstes Screen-Gadget.

sc_DetailPen, sc_BlockPen
Farbtabellennummer des Vorder- (DetailPen) und des Hintergrundes (BlockPen) (übernommen aus NewScreen-Struktur).

sc_SaveColor0
Speicher für die Hintergrundfarbe des Screens (Farbtabellennummer 0). Er wird beim SCREENBEEPING benutzt, um die Farbe zu restaurieren.

***sc_BarLayer**
Zeiger auf die Layer-Struktur, in dem die Titelzeile abgelegt ist.

***sc_ExtData**
Zeiger auf externe Daten für Screen.

***sc_UserData**
Zeiger auf Daten, die vom Benutzer definiert, benutzt und eingebunden werden können (übernommen aus NewScreen-Struktur).

Sicherlich scheinen die vielen Einträge undurchschaubar, doch werden sie im Laufe der folgenden Kapitel immer klarer.

Interessant zu wissen ist, daß die Screens untereinander verkettet sind und jeder Screen auch eine Liste aller Windows, die ihm "gehören", besitzt (*sc_NextScreen, *sc_FirstWindow). So ist die Verwaltung für Intuition einfacher und flexibler.

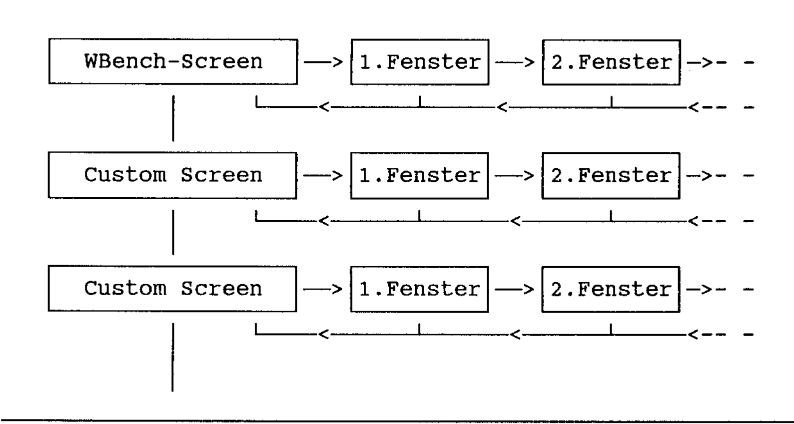


Bild 5.4: Die Verkettung von Screens und Windows

Auch für die Screens gibt es spezielle Funktionen. Einige ausgewählte sind wieder in dem Demonstrations-Programm zu diesem Abschnitt aufgeführt. Die wichtigsten Funktionen sollen hier in alphabetischer Reihenfolge kurz erläutert werden.

CloseWorkBench	=	**-78**	**(Intuition-Library)**

Success **d0** > Mißerfolg = 0

Erklärung Durch CloseWorkbench wird versucht, den Workbench-Screen zu schließen. Dies kann nur funktionieren, wenn sich keine Fenster (oder nur Workbench-Diskverzeichnis-Fenster) darauf befinden. Der zurückgelieferte Wert gibt Aufschluß über den Erfolg (0=Mißerfolg) (siehe auch OpenWorkBench).

DisplayBeep	=	**-96**	**(Intuition-Library)**

***Screen** **a0** < Zeiger auf eine Screen-Struktur.

Erklärung	Die Hintergrundfarbe des angegebenen Screens wird kurzzeitig geändert (Aufblinken).

GetScreenData	=	**-426 (Intuition-Library)**

Buffer	a0	<	Zeiger auf einen Puffer.
***Screen**	a1	<	Zeiger auf eine Screen-Struktur.
Size	d0	<	Größe des angegebenen Puffers in Byte.
Type	d1	<	Typ des Screens (CUSTOM- oder WBENCH-SCREEN)
Success	d0	>	Ist ein Fehler aufgetreten, ist Success mit Null initialisiert.
Erklärung			Mit der Funktion GetScreenData werden die Screen-Daten in den angegebenen Puffer geschrieben.

MakeScreen	=	**-376 (Intuition-Library)**

***Screen**	a0	<	Zeiger auf eine Screen-Struktur.
Erklärung			Die Systemstrukturen zur Verwaltung der Grafikhardware für einen Screen werden (neu) eingerichtet.

MoveScreen	=	**-162 (Intuition-Library)**

***Screen**	a0	<	Zeiger auf eine Screen-Struktur.
dx	d0	<	Wert für horizontale Verschiebung des Screens. Es sind auch negative Werte erlaubt. ACHTUNG: Das horizontale Verschieben eines Screens wird erst ab Kickstart 2.0 unterstützt!
dy	d1	<	Wert für vertikale Verschiebung des Screens. Es sind auch negative Werte erlaubt.
Erklärung			Die Funktion MoveScreen verschiebt den Screen in die angegebene Richtung um einen angegebenen Wert.

OpenWorkBench	=	**-210 (Intuition-Library)**

Screen	d0	>	Zeiger auf die Screen-Struktur der WorkBench.
Erklärung			Es wird versucht, den Workbench-Screen zu öffnen (siehe auch CloseWorkBench).

RemakeDisplay	=	-384 (Intuition-Library)

Erklärung — MakeScreen-Aufruf für alle Screens.

RethinkDisplay	=	-390 (Intuition-Library)

Erklärung — Durch RethinkDisplay werden alle Werte, die für die Screen-Darstellung durch die Grafikhardware verantwortlich sind, überprüft.

ScreenToBack	=	-246 (Intuition-Library)

*Screen	a0	<	Zeiger auf eine Screen-Struktur.

Erklärung — Der angegebene Screen wird in den Hintergrund gestellt (siehe auch ScreenToFront).

ScreenToFront	=	-252 (Intuition-Library)

*Screen	a0	<	Zeiger auf eine Screen-Struktur.

Erklärung — Der angegebene Screen wird in den Vordergrund geholt (siehe auch ScreenToBack).

ShowTitle	=	-282 (Intuition-Library)

*Screen	a0	<	Zeiger auf eine Screen-Struktur.
Show	d0	<	Durch Show wird angegeben, ob der Titel gezeigt (<> 0) oder nicht gezeigt (= 0) werden soll.

Erklärung — Die Titelzeile eines Screens wird aus- oder eingeschaltet.

WBenchToBack	=	-336 (Intuition-Library)

Erklärung — Der Workbench-Screen wird hinter alle anderen Screens plaziert (siehe auch WBenchToFront).

WBenchToFront	=	-342 (Intuition-Library)

Erklärung Der Workbench-Screen wird im Vordergrund dargestellt (siehe auch WBenchToBack).

5.3 Die Grafikstrukturen

Bevor wir uns an die Gadgets wagen, sollten wir uns drei interessante Strukturen ansehen: die IntuiText-, die Border- und die Image-Struktur. Alle drei können oder müssen in die Gadget- bzw. Menü-Struktur eingebunden werden; man kann die Strukturen jedoch auch getrennt benutzen. Sie dienen alle dazu, etwas auszugeben, nämlich einen Text, einen Rahmen oder ein Bild.

5.3.1 IntuiText-Struktur

Den Anfang soll die IntuiText-Struktur machen. Sie enthält neben dem Text selber noch weitere Informationen, wie z.B. die Position und die Farbe des Textes. Natürlich brauchen wir, um einen IntuiText auszugeben, wieder eine Intuition-Funktion. Ihr Name ist PrintIText und sie erwartet vier Parameter:

PrintIText	=	-216 (Intuition-Library)

***RPort**	**a0**	<	Zeiger auf einen RastPort
***IText**	**a1**	<	Zeiger auf eine IntuiText-Struktur
LeftOffset	**d0**	<	X-Koordinate relativ zur angegebenen X-Koordinate.
TopOffset	**d1**	<	Y-Koordinate relativ zur angegebenen Y-Koordinate.

Erklärung Die Funktion PrintIText gibt den durch eine IntuiText-Struktur angegebenen Text auf dem angegebenen RastPort an der angegebenen Position aus.

In a1 wird ein Zeiger auf eine IntuiText-Struktur, dessen Text ausgegeben werden soll, geladen. Die Datenregister d0 und d1 enthalten die Position des Textes und in a0 wird ein Zeiger auf den RastPort, in dem der Text ausgegeben werden soll, gelegt.

Nun, was ist ein solcher RastPort? Der RastPort ist eine Struktur, die wichtige Daten über einen Bereich enthält, auf den man grafisch zugreifen kann. Dazu gehören Informationen über den zu benutzenden Zeichensatz, den zuletzt gewählten Zeichenmodus, die Farbe und vieles mehr. Dieser RastPort wurde von Intuition erstellt, als es unser Fenster geöffnet

hat. Der Zeiger auf diese wichtige Stuktur wurde in die Window-Struktur eingetragen, auf die wir mittels des zurückgelieferten Pointers zugreifen können. Man liest den Wert über einen Offset, den man der Window-Struktur entnehmen kann, aus. Den Zeiger finden wir ab dem 50. Byte vom Strukturanfang:

```
        ...
        move.l   WindowHD,a0     ; RastPort-Adresse auslesen
        move.l   50(a0),RPort
        ...
```

Haben wir den Zeiger auf den RastPort unseres Fensters, können wir uns den Funktionsaufruf der PrintIFunktion ansehen:

```
PrintIText    =       -216

        ...
        move.l   RPort,a0        ; Zeiger auf RastPort
        lea      IText0,a1       ; Zeiger auf IntuiText
        move.l   #30,d0          ; Position
        move.l   #42,d1
        jsr      PrintIText(a6) ; IntuiText ausgeben
        ...
```

Das einzige Element das uns noch fehlt, ist die IntuiText-Struktur, ohne die wir keinen Text ausgeben können.

IntuiText-Struktur:

```
00     dc.b     it_FrontPen         ; Farben
01     dc.b     it_BackPen          ;
02     dc.b     it_DrawMode         ; Zeichenmodus
03     dc.b     it_KludgeFill00     ; Füllbyte
04     dc.w     it_LeftEdge         ; relative X-Koordinate
06     dc.w     it_TopEdge          ; relative Y-Koordinate
08     dc.l    *it_ITextFont        ; TextAttr-Struktur
12     dc.l    *it_IText            ; Zeiger auf Zeichenkette
16     dc.l    *it_NextIText        ; nächste IntuiText-Struktur
20              it_SIZEOF
```

it_FrontPen, it_BackPen
Farbtabellennummer der Vordergrund- und Hintergrundfarbe.

it_DrawMode
Zeichenmodus, der zur Ausgabe des Textes benutzt werden soll. Hier kann man zwischen einigen Einstellungen wählen:

Modus	Wert	Bedeutung
dJAM1	0	Die Farbe wird für den Vordergrund benutzt.
JAM2	1	Ein nicht gesetztes Bit in der auszugebenen Grafik wird mit der eingestellten Hintergrundfarbe gezeichnet.
COMPLEMENT	2	Die Grafik wir durch die XOR-Funktion

		eingetragen.
INVERSVID	4	Die Ausgabe wird invertiert

Natürlich sind auch Kombinationen der Flags möglich.

it_LeftEdge, it_TopEdge
Positionierung des Textes relativ zu den in den Datenregistern angegebenen Koordinaten.

***it_ITextFont**
Zeiger auf den zu benutzenden Zeichensatz oder eine Null, wenn der Standardzeichensatz benutzt werden soll.

***it_IText**
Zeiger auf die Zeichenkette, in welcher der Text steht, der ausgegeben werden soll. Die Zeichenkette muß mit Null abgeschlossen sein!

***it_NextText**
Adresse einer weiteren IntuiText-Struktur, die ausgegeben werden soll. Durch diese Verkettung kann man mehrer Texte mit einem Funktionsaufruf ausgeben lassen. Soll kein weiterer Text ausgegeben werden, setzt man den Wert auf Null.

Das war die IntuiText-Struktur. Das Demo-Programm für diese Funktion sparen wir uns bis zur Image-Struktur auf. Dann werden alle drei Grafikfunktionen in einem Programm demonstriert.

IntuiTextLength	=	**-330 (Intuition-Library)**

***IText**	**a0**	<	Zeiger auf die IntuiText-Struktur dessen Länge errechnet werden soll.
Len	**d0**	>	Länge der Zeichenkette in Pixeln, die in der angegebenen IntuiText-Struktur enthalten ist.
Erklärung			Mit Hilfe dieser Funktion ist es möglich, die Länge eines IntuiTextes berechnen zu lassen. Die Länge ist abhängig von dem ITextFont, der verwendet wird.

5.3.2 Border-Struktur

Nun zur Border-Struktur! Ein Border ist eine Umrandung, die aus einzelnen Linien besteht, die zwischen angegebenen Punkten gezogen werden. Man benutzt sie z.B., um Gadgets und Text zu umranden oder einfache Zeichnungen auf einem RastPort auszugegeben. Auch hierzu stellt Intuition eine Funk-

tion bereit, mit der man einen Rahmen, unabhängig von Gadgets, zeichnen kann. Sie heißt DrawBorder und benötigt wie die PrintIText-Funktion einen Zeiger auf den RastPort, auf den gezeichnet werden soll, und natürlich einen Zeiger auf die Border-Struktur sowie die Koordinaten, an denen der Rahmen gezeichnet werden soll.

DrawBorder	=	**-108 (Intuition-Library)**

***RPort**	**a0**	<	Zeiger auf einen RastPort.
***Border**	**a1**	<	Zeiger auf eine Border-Struktur.
LeftOffset	**d0**	<	X-Position relativ zur angegebenen X-Koordinate.
TopOffset	**d1**	<	Y-Position relativ zur angegebenen Y-Koordinate.

Erklärung DrawBorder zeichnet den angegebenen Rahmen auf dem übergebenen RastPort.

Der Aufruf im Programm sieht folgendermaßen aus:

```
DrawBorder    =       -108

        ...
        move.l  RPort,a0
        lea     Border,a1       ; Zeiger auf Border-Struktur
        move.l  #20,d0          ; Position
        move.l  #30,d1
        jsr     DrawBorder(a6)  ; Border ausgeben
        ...
```

Bild 5.5: Zeichnen eines Borders

Jetzt, wo wir wissen, wie wir die DrawBorder-Funktion aufrufen können, müssen wir uns den Aufbau der Border-Struktur ansehen.

Border-Struktur:

```
00      dc.w     bd_LeftEdge          ; X-Koordinate
02      dc.w     bd_TopEdge           ; Y-Koordinate
04      dc.b     bd_FrontPen          ; Vordergrund
05      dc.b     bd_BackPen           ; Hintergrund
06      dc.b     bd_DrawMode          ; Zeichenmodus
07      dc.b     bd_Count             ; Anzahl der XY-Paare
08      dc.l    *bd_XY                ; Zeiger auf XY-Paare
12      dc.l    *bd_NextBorder        ; Zeiger auf nächsten Border
16               bd_SIZEOF
```

bd_LeftEdge, bd_TopEdge
Position des Borders relativ zu den in den Datenregistern übergebenen Koordinaten.

bd_FrontPen, bd_BackPen
Farbtabellennummer der Vorder- und Hintergrundfarbe.

bd_DrawMode
Zeichenmodus, in dem der Border gezeichnet werden soll.

bd_Count
Anzahl der XY-Paare, die mit Linien verbunden werden sollen.

***bd_XY**
Zeiger auf ein Wort-Array mit den XY-Koordinaten der Border-Punkte.

***bd_NextBorder**
Adresse der nächsten Border-Struktur, die gezeichnet werden soll.

Das Demonstrationsprogramm hierzu finden sie am Ende des nächsten Kapitels.

5.3.3 Image-Struktur

Nachdem wir die IntuiText- und die Border-Strukturen besprochen haben, fehlt nur noch die Image-Struktur, die eine Grafikstruktur ist. Um ein solches Image (Bildnis) auszudrukken, benötigen wir die Funktion DrawImage.

DrawImage	=	**-114 (Intuition-Library)**

***RPort**	**a0**	<	Zeiger auf den RastPort.
***Image**	**a1**	<	Zeiger auf die Image-Struktur.
LeftOffset	**d0**	<	X-Koordinate.
TopOffset	**d1**	<	Y-Koordinate.

Erklärung DrawImage gibt die angegebenen Image-Daten auf den angegebenen RastPort aus.

Die Parameter sind eigentlich die gleichen wie bei den vorangegangenen Funktionen. Der einzige Unterschied ist, daß in a1 ein Zeiger auf eine Image-Struktur stehen muß. Auch der Aufruf im Programm sieht ähnlich aus:

```
DrawImage     =       -114
        ...
        move.l   RPort,a0
        lea      Image,a1        ; Zeiger auf Image-Struktur
        move.l   #230,d0         ; Position
        move.l   #36,d1
        jsr      DrawImage(a6)   ; Image zeichnen
        ...
```

Bild 5.6: Zeichnen eines Image

Wichtig ist noch zu wissen, aus welchen Einträgen sich die Image-Struktur zusammensetzt. Das wollen wir uns sofort ansehen:

Image-Struktur:

```
00    dc.w    ig_LeftEdge        ; X-Koordinate
02    dc.w    ig_TopEdge         ; Y-Koordinate
04    dc.w    ig_Width           ; Breite des Images
06    dc.w    ig_Height          ; Höhe des Images
08    dc.w    ig_Depth           ; Tiefe (Anzahl der BitMaps)
10    dc.l   *ig_ImageData       ; Zeiger auf ImageData
14    dc.b    ig_PlanePick       ; PlanePick-Daten
15    dc.b    ig_PlaneOnOff      ; PlaneOnOff-Daten
16    dc.l   *ig_NextImage       ; Zeiger auf nächstes Image
20            ig_SIZEOF
```

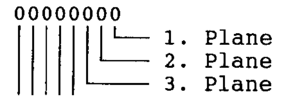
ig_LeftEdge, ig_TopEdge
Position des Images relativ zu den angegebenen Koordinaten.

ig_Width, ig_Height
Breite und Höhe der Grafikdaten.

ig_Depth
Anzahl der BitPlanes, aus dem sich das Bild zusammensetzt. Dadurch ist auch gleich die Anzahl der verwendeten Farben festgelegt. (2^{Depth} (Tiefe) = Anzahl der Farben).

***ig_ImageData**
Zeiger auf die Grafikdaten. (Näheres dazu gleich).

ig_PlanePick
Mit diesem 1 Byte großen Wert kann man einstellen, welche BitPlanes gezeichnet werden sollen. Dazu kann man mittels der einzelnen Bits jede Plane an- und ausschalten.



Durch die jeweilige Selektion der BitPlanes kann man das Bild in verschiedenen Formen darstellen.

ig_PlaneOnOff
Auch in diesem Eintrag kann man alle Planes einzeln an- und ausschalten. Jede Plane, die auf diese Weise angeschaltet ist, wird in der zugehörigen Window-Plane auf eins gesetzt. Die ursprünglichen Grafikdaten dieser Plane werden nicht benutzt. Dadurch ist es möglich, die Grafik in einer anderen Farbe darzustellen.

***ig_NextImage**
Zeiger auf die nächste Image-Struktur, die gezeichnet werden soll, oder eine Null.

Das war die gesamte Image-Struktur. Nur einen Eintrag sollten wir an dieser Stelle noch etwas genauer unter die Lupe nehmen. Das ist der Zeiger auf die Daten der Grafik (*ig_ImageData). Um die Zusammensetzung der Daten zu verstehen, müssen wir etwas weiter ausholen und uns den Aufbau einer Computergrafik vor Augen führen.

Eine Computergrafik ist eigentlich nichts anderes als eine Anzahl von Punkten, die verschiedene Farben (oder Helligkeitswerte) haben und zusammengesetzt ein Bild ergeben. Sicherlich haben sie dies schon festgestellt, als sie den Mauszeiger mit Preferences bearbeitet haben.

Da die Werte der Farben in einer Tabelle definiert worden sind, braucht man nur die Farbtabellennummer für einen Punkt zu bestimmen. Diese Nummern werden in sogenannten BitPlanes kodiert abgelegt. Wollen wir nun eine zweifarbige Grafik erzeugen, dann symbolisiert jedes Bit einer BitPlane einen Punkt. Man hat $2^1 = 2$ Farben zur Verfügung. Die gewählte Farbe wird durch den Zustand des Bits festgelegt (0 = Farbe 0 oder 1 = Farbe 1). So entstehen zweifarbige Bilder.

Beispiel für eine Grafik mit zwei Farben:

```
dc.l   %10010001000100001000001110001000
dc.l   %10010010100100001000010001001000
dc.l   %11110100010100001000010001001000
dc.l   %10010111110100001000010001000000
dc.l   %10010100010111101111001110001000
```

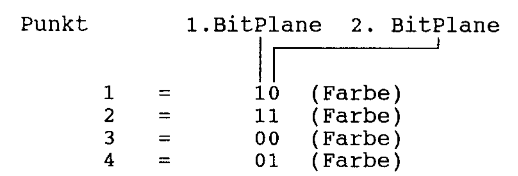
(Die Einsen sind fettgedruckt, damit die Grafik besser zu erkennen ist.)

Mit zwei Farben ist alles noch ganz einfach. Will man jetzt aber mehr Farben verwenden, muß man auch mehr BitPlanes benutzen. Daraus ergibt sich, daß für einen Punkt jetzt nicht mehr ein Bit sondern zwei oder mehr verantwortlich sind.

Die Nummer der Farbe des ersten Punktes wird dann durch die Kombination des ersten Bits der ersten BitPlane und des ersten Bits der zweiten Plane festgelegt. Sie werden quasi übereinander gelegt.

```
   1.BP  ┌──────────────────────────────────┐
         │11000001000100001000001110001000  │ 1.Langwort
2.BP  ┌──────────────────────────────────┐00│ 2.Langwort
      │01010001000000000100000111000100 0│00│ 3.Langwort
      │00000010100000000100001000100100 0│00│ 4.Langwort
      │00000100010000000100001000100100 0│00│ 5.Langwort
      │00000111110000000100001000100000 0├──┘
      │00000100010000000111100111000100 0│
      └──────────────────────────────────┘
```

Die Bits der ersten vier Punkte zeigen jede mögliche Kombination. Also werden mit den ersten vier Punkten alle vier Farben gezeigt.



Jetzt wird auch klar, wie die Anzahl der Farben festgelegt ist. Weil jedes Bit nur zwei Werte annehmen kann, ist die Rechnung $2^{(\text{Anzahl der BitPlanes})}$ = (Anzahl der Farben) verständlich. Jede Grafik, die auf dem Amiga zu sehen ist, ist im Speicher in diese binäre Notierung zerlegt.

Nach diesem kleinen Exkurs in die Grafikwelt des Amiga kommen wir zurück zu unserer Image-Struktur und den ImageDaten, die wir noch benötigen.

```
            Section "",Data_C
ImageData:

            dc.l    %11111111111111111111111111111111 ; Plane 1
            dc.l    %10000000000000000000000000000000
            dc.l    %10000000000000000000000000001000
            dc.l    %10000000000000000000000000001000
            dc.l    %10000000000000000000000000001000
            dc.l    %10000000001111111111100000001000
            dc.l    %10000000001111111111100000001000
            dc.l    %10000000001111111111100000001000
            dc.l    %10000000001111111111100000001000
            dc.l    %10000000001111111111100000001000
            dc.l    %10000000001111111111100000001000
            dc.l    %10000000001000000000000000001000
            dc.l    %10000000000000000000000000001000
            dc.l    %10000000000000000000000000001000
            dc.l    %10000000000000000000000000001000
            dc.l    %10001111111111111111111111111000
            dc.l    %10000000000000000000000000000000
            dc.l    %10000000000000000000000000000000

            dc.l    %00000000000000000000000000000000 ; Plane 2
            dc.l    %00000000000000000000000000000001
            dc.l    %00011111111111111111111111111001
            dc.l    %00010000000000000000000000000001
            dc.l    %00010000000000000000000000000001
            dc.l    %00010000000000000000000000000001
            dc.l    %00010000000111111111110000000001
            dc.l    %00010000000111111111110000000001
            dc.l    %00010000000111111111110000000001
            dc.l    %00010000000111111111110000000001
```

```
            dc.l    %00010000000111111111110000000001
            dc.l    %00010000000111111111110000000001
            dc.l    %00010000000111111111110000000001
            dc.l    %00010000000000000000000000000001
            dc.l    %00010000000000000000000000000001
            dc.l    %00010000000000000000000000000001
            dc.l    %00000000000000000000000000000001
            dc.l    %01111111111111111111111111111111
```

Bild 5.7: Definition einer Image-Grafik

Abgesehen von dem recht einfallslosen Bild, ist Ihnen sicher die Zeile "Section '',Data_C" aufgefallen. Dies ist eine vom DevPac-Assembler benutzte Direktive. Sie bewirkt, daß die Daten ab diesem Eintrag in das Chip-RAM geladen werden. Diese Einstellung ist unbedingt notwendig, da es zwei unterschiedliche Bereiche des Speichers gibt: den Chip- und den Fast-RAM-Bereich. Letzterer kann nur vom Prozessor benutzt werden und ist daher schneller. Das Chip-RAM ist dagegen sowohl vom Prozessor als auch von den Customchips nutzbar. Diese Customchips sind Zusatzprozessoren, die unter anderem auch für die Ausgabe der Grafik zuständig sind. Darum müssen die Grafikdaten, die wir ausgeben wollen, unbedingt im Chip-RAM stehen, da sonst die Customchips keinen Zugriff hätten. Die Folge wäre Grafikmüll. Auch wenn man nur 512 KB (also nur Chip-RAM) hat, sollte man diese Einstellung vornehmen, um die Kompatibilität zu anderen Rechnern aufrechtzuhalten.

Sollte der Assembler diese oder eine ähnliche Einstellung nicht unterstützen, dann benutzen sie einfach das Tool NO-FASTMEM. Es schaltet den Fast-RAM-Bereich ab und zwingt den Rechner, alle Daten ins Chip-RAM zu laden.

Man kann aber davon ausgehen, daß jeder gebräuchliche Assembler eine solche Funktion anbietet. Der Seka-Assembler z.B. stellt zu Beginn den Speicherbereich, den man benutzen will, zur Auswahl.

Abschließend folgt nun das versprochene Demo-Programm. Zunächst wird ein Fenster auf der Workbench geöffnet und die Grafiken und Texte ausgegeben. Durch das Close-Gadget kann man das Programm wie immer beenden.

```
* Programm 5.9: Anwendung der Grafik-Strukturen

ExecBase        =       4
OldOpenLib      =       -408
CloseLib        =       -414
GetMsg          =       -372
ReplyMsg        =       -378
WaitPort        =       -384
OpenWindow      =       -204
```

```
CloseWindow    =        -72
PrintIText     =        -216
DrawBorder     =        -108
DrawImage      =        -114


Start:
        move.l  ExecBase,a6     ; Intuition-Library öffnen
        lea     IntName,a1
        jsr     OldOpenLib(a6)
        move.l  d0,IntBase
        beq     IntError

        move.l  IntBase,a6      ; Fenster öffnen
        lea     WindowArgs,a0
        jsr     OpenWindow(a6)
        move.l  d0,WindowHD
        beq     WinError

        move.l  WindowHD,a0     ; Zeiger auf User- und
        move.l  86(a0),UPort    ; RastPort speichern
        move.l  50(a0),RPort

        move.l  RPort,a0        ; Zeiger auf RastPort
        lea     IText0,a1       ; Zeiger auf IntuiText
        move.l  #30,d0          ; Position
        move.l  #42,d1
        jsr     PrintIText(a6)  ; IntuiText ausgeben

        move.l  RPort,a0
        lea     Border,a1       ; Zeiger auf Border-Struktur
        move.l  #20,d0          ; Position
        move.l  #30,d1
        jsr     DrawBorder(a6)  ; Border ausgeben

        move.l  RPort,a0
        lea     Image,a1        ; Zeiger auf Image-Struktur
        move.l  #230,d0         ; Position
        move.l  #36,d1
        jsr     DrawImage(a6)   ; Image zeichnen

MessageLoop:
        move.l  ExecBase,a6

        move.l  UPort,a0
        jsr     GetMsg(a6)      ; Nachricht vom UPort holen
        move.l  d0,Message
        bne     MessageBranch   ; War was ?

        move.l  UPort,a0
        jsr     WaitPort(a6)    ; Nein, dann warten
        bra     MessageLoop

Exit:
        move.l  IntBase,a6      ; Window schließen
        move.l  WindowHD,a0
```

```
        jsr      CloseWindow(a6)

WinError:
        move.l   ExecBase,a6     ; Intuition-Library schließen
        move.l   IntBase,a1
        jsr      CloseLib(a6)

IntError:
        rts                      ; Ende !


MessageBranch:
        move.l   Message,a0      ; IDCMP-Flag aus Class
        move.l   20(a0),Class    ; Eintrag lesen

        move.l   Message,a1      ; Nachricht bestätigen
        jsr      ReplyMsg(a6)

        cmp.l    #$200,Class     ; Close-Gadgets betätigt ?
        beq      Exit            ; Ja, dann zum Ende springen

        bra      MessageLoop

* Datenbereich

Class:        dc.l    0              ; Class Eintrag
Message:      dc.l    0              ; Zeiger auf IntuiMessage
UPort:        dc.l    0              ; Zeiger auf UserPort
RPort:        dc.l    0              ; Zeiger auf RastPort
IntBase:      dc.l    0              ; Adresse der Intuition-Lib
WindowHD:     dc.l    0              ; Zeiger auf Window-Struktur

IntName:      dc.b    "intuition.library",0
              even

WinName:      dc.b    "Close-Gadget beendet das Programm !",0
              even

WindowArgs:                          ; Definitionene des Fensters
              dc.w    150,50,340,90
              dc.b    1,3
              dc.l    $200,$1100F,0,0,WinName
ScreenHD:     dc.l    0,0
              dc.w    100,50,200,100,1

* IntuiText-Struktur

IText0:       dc.b    1,3            ; Farben
              dc.b    0              ; Zeichenmodus
              even                   ; (Füllbyte)
              dc.w    0,0            ; relative Position
              dc.l    0              ; Zeiger auf Fontsatz
              dc.l    ITextData0     ; Zeiger auf Zeichenkette
              dc.l    IText1         ; Zeiger auf nächsten IntuiText

ITextData0:   dc.b    "Tolles Bild oder? =>",0
```

```
            even

IText1:     dc.b    2,3             ; Definitionen der zweiten
            dc.b    0               ; IntuiText-Struktur
            even
            dc.w    -1,-1
            dc.l    0
            dc.l    ITextData1
            dc.l    0

ITextData1: dc.b    "Tolles Bild oder? =>",0
            even

* Border-Data

Border:     dc.w    0,0             ; Positionierung
            dc.b    2,0             ; Farben
            dc.b    0,5             ; Zeichenm./Anzahl der XY-Paare
            dc.l    BorderData      ; Zeiger auf XY-Paare
            dc.l    Border0         ; Zeiger auf nächsten Rahmen

BorderData:                         ; Borderdaten
            dc.w    0,0             ; XY-Koordinaten der 5 Punkte
            dc.w    270,0
            dc.w    270,30
            dc.w    0,30
            dc.w    0,0

Border0:    dc.w    1,1             ; Definition der zweiten
            dc.b    1,0,0,3         ; Border-Struktur
            dc.l    BorderData0,0

BorderData0: dc.w   270,0
            dc.w    270,30
            dc.w    2,30

* Image-Data

Image:      dc.w    0,0             ; Position
            dc.w    32,20           ; Breite und Höhe der Daten
            dc.w    2               ; Tiefe
            dc.l    ImageData       ; Zeiger auf Daten
            dc.b    %11,0           ; PlanePick/PlaneOnOff
            dc.l    0               ; Zeiger auf nächstes Image

            Section "",Data_C       ; Direktive um die Grafikdaten ins
                                    ; CHIP-RAM zu legen

ImageData:

            dc.l    $FFFFFFFF,$80000000,$80000008,$80000008 ; Pl1
            dc.l    $80000008,$80000008,$803FFC08,$803FF808
            dc.l    $803FF808,$803FF808,$803FF808,$803FF808
            dc.l    $80200008,$80000008,$80000008,$80000008
            dc.l    $80000008,$8FFFFFF8,$80000000,$80000000
```

```
            dc.l    $00000000,$00000001,$1FFFFFF9,$10000001 ;P12
            dc.l    $10000001,$10000001,$10000001,$101FFC01
            dc.l    $101FFC01,$101FFC01,$101FFC01,$101FFC01
            dc.l    $101FFC01,$103FFC01,$10000001,$10000001
            dc.l    $10000001,$10000001,$00000001,$FFFFFFFF
```

Programm 5.9: Anwendung der Grafik-Strukturen

5.4 Gadgets

Mit diesem reichhaltigen Wissen über die Grafik-Strukturen können wir uns an die Gadgets wagen. Sie bieten eine sehr einfache aber komfortable Möglichkeit der Kommunikation zwischen dem Benutzer und dem Rechner. Ein Gadget (Apparat/Vorrichtung) hat zahlreiche Erscheinungsformen und Merkmale, die wie schon bei den Windows und Screens in einer Struktur zusammengefaßt werden. Intuition stellt uns drei unterschiedliche Typen zur Verfügung:

Boolean-Gadgets — Die Boolean-Gadgets kann man als eine Art Schalter oder Taster benutzen. Dabei muß der Benutzer mit der Maus das Gadget anklicken.

String-/Integer-Gadgets — Ein String-Gadget bietet die Möglichkeit, an einer bestimmten Stelle des Fensters einen Text einzugeben. Integer-Gadgets gehören auch zur Art der String-Gadgets. Sie dienen jedoch dazu, Zahlen einzugeben.

Proportional-Gadgets — Der dritte und letzte Typ der Gadgets heißt Proportional-Gadget. Er besteht aus einen Schieber, den man in einem Rahmen beliebig positionieren kann. Dadurch wird es möglich, einen Wert einzustellen, der proportional zur Schieberposition im Rahmen ist.

Anfangen wollen wir mit dem Boolean-Gadget. Es ist das einfachste Gadget und soll uns dabei helfen, die Grundstruktur der Gadgets kennenzulernen.

An dieser Stelle möchten wir darauf hinweisen, daß wir darauf verzichten wollen, die umfangreichen Demonstrationsprogramme abzudrucken, da sie alle auf der beiliegenden Diskette zu finden sind.

5.4.1 Boolean-Gadget

Ein Boolean-Gadget kann ein Schalter oder auch ein Taster sein. Es ist wohl das am häufigsten benutzte Gadget. Wie bei fast allen Einrichtungen des Betriebssystems müssen wir natürlich auch für die Gadgets eine Struktur anlegen. Dabei gibt es eine Grundstruktur, die von allen drei Gadgettypen benutzt wird. Ihren Aufbau wollen wir jetzt untersuchen. Bei den beiden übrigen Gadgettypen werden wir auf diese Struktur nicht mehr gesondert eingehen.

Gadget-Struktur:

```
00    dc.l   *gg_NextGadget       ; Zeiger auf nächstes Gadget
04    dc.w    gg_LeftEdge         ; X-Position
06    dc.w    gg_TopEdge          ; Y-Position
08    dc.w    gg_Width            ; Breite des Gadgets
10    dc.w    gg_Height           ; Höhe des Gadgets
12    dc.w    gg_Flags            ; Gadget-Flags
14    dc.w    gg_Activation       ; Activation-Flags
16    dc.w    gg_GadgetType       ; Gadgettyp
18    dc.l   *gg_GadgetRender     ; "normale" Datenstruktur
22    dc.l   *gg_SelectRender     ; "aktivierte" Datenstruktur
26    dc.l   *gg_GadgetText       ; Zeiger auf IntuiText
30    dc.l    gg_MutualExclude    ; MutualExclude-Daten
34    dc.l    gg_SpecialInfo      ; SpecialInfo
38    dc.w    gg_GadgetID         ; Gadgetidentifikationsnummer
40    dc.l    gg_User             ; User-Daten
44            gg_SIZEOF
```

***gg_NextGadget**
Zeiger auf das nächste Gadget. Da die Gadgets in einer einfachen Liste verwaltet werden, sind sie durch Zeiger verknüpft. Hierzu dient das erste Langwort. Will man weitere Gadgets einbinden, so setzt man hier die Adresse der nächsten Struktur ein. Das Listenende ist erreicht, wenn der Zeiger mit Null initialisiert worden ist. Im Gegnsatz zu den Screens und Windows, bei denen eine neue Struktur angelegt wird, wird hier die vom Programmierer definierte Struktur auch von Intuition benutzt. Deshalb sollte man die Einträge nicht willkürlich überschreiben.

gg_LeftEdge, gg_TopEdge, gg_Width, gg_Hight
Position und Größe der Hit-Box. Das ist der Bereich, in dem der Mauszeiger (bei gedrückter bzw. losgelassener linker Maustaste) eine Meldung an unser Programm verursacht.

gg_Flags
Durch die Flags ergeben sich zahlreiche Einstellungsmöglichkeiten. Die einzelnen Flags haben dabei folgende Bedeutungen:

Gadget-Flag	Wert	Bedeutung
GADGHCOMP	$0003	Hit-Box bei Aktivierung invertieren
GADGHBOX	$0001	Hit-Box wird umrandet
GADGHIMAGE	$0002	SelectRender-Grafik wird angezeigt
GADGHNONE	$0003	keine Reaktion
GRELBOTTOM	$0008	Y-Position relativ zum Boden
GRELRIGHT	$0010	X-Position relativ zur rechten Seite
GRELWIDTH	$0020	Breite proportional zum Fenster
GRELHEIGHT	$0040	Höhe proportional zum Fenster
SELECTED	$0080	Gadget aktiviert
GADGDISABLED	$0100	Gadget nicht auswählbar
GADGIMAGE	$0004	GadgetRender und SelectRender sind Zeiger auf Image-Strukturen

Die ersten vier Flags geben an, wie das Gadget reagieren soll, wenn es aktiviert worden ist.

GADGHCOMP Inhalt der Hit-Box invertiert darstellen.

GADGHBOX Das Gadget mit einem Rahmen umgeben.

GADGHIMAGE Anstelle der Grafik GadgetRender wird die Grafik SelectRender gezeichnet.

GADGHNONE Keine Reaktion.

Die nächsten Flags bestimmen, wie die Einträge LeftEdge, TopEdge, Width und Height interpretiert werden sollen. Dabei kann man z.B. die Größe eines Gadget abhängig von der Fenstergröße machen.

GRELBOTTOM Der gg_TopEdge-Eintrag wird als relative Position zum unteren Rand des Fensters gewertet.

GRELRIGHT Der gg_LeftEdge-Eintrag bestimmt wird als realtive Position zum rechten Rand des Fensters gewertet.

GRELWIDTH Der Wert gg_Width bestimmt den Abstand des linken Gadgetrandes vom rechten Windowrand. Dadurch paßt sich das Gadget automatisch an jede Größenveränderung des Fensters an.

GRELHEIGHT Der Wert gg_Height bestimmt den Abstand des unteren Gadgetrandes vom unteren Windowrand. Dadurch paßt sich das Gadget automatisch an jede Größenveränderung des Fensters an.

Als letztes kommen nun die restlichen Flags:

SELECTED Das Gadget ist direkt aktiviert, wenn es gezeichnet wird. Dieses Flag kann zu Beginn gesetzt werden und später nur noch gelesen werden (siehe auch ActivateGadget).

GADGDISABLED Das Gadget ist nicht wählbar, die Hit-Box wird mit einem Punktmuster

	überzogen. Dieses Flag kann auch nur am Anfang gesetzt werden (siehe auch GadgetOn und GadgetOff).
GADGIMAGE	Die Einträge GadgetRender und SelectRender werden als Zeiger auf Image-Strukturen (sonst Border-Strukturen) interpretiert.

gg_Activation
Die Activation-Flags bestimmen, wann das Gadget aktiviert ist und was passieren soll. Auch zu diesen Einstellungen eine Tabelle, um die Möglichkeiten, die Intuition zur Verfügung stellt, kennenzulernen.

Activation-Flag	Wert	Bedeutung
TOGGLESELECT	$0100	Schalter-Gadget statt Taster-Gadget
RELVERIFY	$0001	Taste wurde über Gadget losgelassen
GADGIMMEDIATE	$0002	Gadget wird direkt aktiviert
RIGHTBORDER	$0010	Gadget in rechten Rand des Fensters
LEFTBORDER	$0020	Gadget in linken Rand des Fensters
TOPBORDER	$0040	Gadget in oberen Rand des Fensters
BOTTOMBORDER	$0080	Gadget in unteren Rand des Fensters
STRINGCENTER	$0200	Zeichenkette zentrieren
STRINGRIGHT	$0400	Zeichenkette rechtsbündig
LONGINT	$0800	String-Gadget zu Integer-Gadget
ALTKEYMAP	$1000	Tastaturtabelle auf Eingabe anwenden
BOOLEXTEND	$2000	Zusatzstruktur für Boolean-Gadgets
ENDGADGET	$0004	ENDGADGET des Requesters
FOLLOWMOUSE	$0008	Mausposition wird gemeldet

TOGGLESELECT	Das Gadget wird zu einem Schalter umgewandelt und dreht bei Selektion seinen Zustand um.
RELVERIFY	Gadget wird erst aktiviert, wenn der linke Mausknopf auch über dem Bereich der Hit-Box wieder losgelassen wird.
GADGIMMEDIATE	Das Gadget ist sofort aktiviert, wenn es angeklickt wurde.
RIGHTBORDER	Gadget in den rechten Rand des Fensters einbinden.
LEFTBORDER	Gadget in den linken Rand des Fensters einbinden.
TOPBORDER	Gadget in den oberen Rand des Fensters einbinden.
BOTTOMBORDER	Gadget in den unteren Rand des Fensters einbinden
STRINGCENTER	Bei einem String-Gadget wird der Text zentriert dargestellt
STRINGRIGHT	Text rechtsbündig im Kontainer des String-Gadgets ausgeben (wird weder STRINGCENTER noch STRINGRIGHT angegeben, so wird der Text linksbündig ausgegeben).

LONGINT	Wandelt ein String-Gadget in ein Integer-Gadget, welches nur die Eingabe von Zahlen erlaubt.
ALTKEYMAP	Die Eingabe eines String- oder Integer-Gadgets wird erst mit einer speziellen Tastaturtabelle behandelt
BOOLEXTEND	Im Eintrag SpecialInfo wird ein Zeiger auf eine BoolInfo-Struktur benötigt. Diese Struktur ist nicht unbedingt notwendig !!!
ENDGADGET	Das Gadget ist als Requester-CLOSE-Gadget ausgelegt, welches den Requester bei Betätigung schließt.
FOLLOWMOUSE	Solange dieses Gadget aktiviert ist, wird dem Programm die Mausposition ständig gemeldet.

gg_GadgetType
Der Gadgettyp wird mittels der folgenden Flags bestimmt:

Gadgettyp-Flag	Wert	Bedeutung
BOOLGADGET	$0001	Boolean-Gadget
GADGET0002	$0002	(noch nicht benutzt)
PROPGADGET	$0003	Proportional-Gadget
STRGADGET	$0004	String-Gadget
SYSGADGET	$8000	System-Gadgets
SCRGADGET	$4000	Screen-Gadget
GZZGADGET	$2000	Gadget für GZZ-Window
REQGADGET	$1000	Requester-Gadget
SIZING	$0010	Size-Gadget
WDRAGGING	$0020	Gadget für Window Verschiebung
SDRAGGING	$0030	Gadget für Screen Verschiebung
WDOWNBACK	$0060	Gadget WindowToBack
SDOWNBACK	$0070	Gadget ScreenToBack
WUPFRONT	$0040	Gadget WindowToFront
SUPFRONT	$0050	Gadget ScreenToFront
CLOSE	$0080	Close-Gadget

BOOLGADGET	Gadget soll vom Typ Boolean sein.
GADGET0002	(Noch unbenutzt)
PROPGADGET	Gadget soll ein Proportional Gadget sein (wird gleich noch erklärt).
STRGADGET	Gadget soll ein String- oder Integer-Gadget sein (wird gleich noch erklärt).

Nun folgen drei Flags, die bestimmen, wo das Gadget eingesetzt werden soll.

SYSGADGET	Das Gadget ist ein System-Gadgets
SCRGADGET	Das Gadget ist ein **SCR**reen**GADGET**.
GZZGADGET	Das Gadget soll im Rand eines GIMMEZEROZERO-Windows erscheinen.
REQGADGET	Das Gadget erscheint in einem Requester

Nun folgen Flags, welche hauptsächlich für das System von Bedeutung sind.

SIZING	Size-Gadget für Windows
WDRAGGING	Gadget für Verschiebung von Windows
SDRAGGING	Gadget für Verschiebung von Screens
WUPFRONT	Gadget WindowToFront
WDOWNBACK	Gadget WindowToBack
SUPFRONT	Gadget ScreenToFront
SDOWNBACK	Gadget ScreenToBack
CLOSE	Close-Gadget (Windows)

***gg_GadgetRender**
Zeiger auf eine Grafikstruktur. Der Typ der Grafikstruktur kann im Eintrag Flags bestimmt werden. Zur Verfügung stehen die Image- und Border-Struktur.

***gg_SelectRender**
Wie bei GadgetRender wird ein Zeiger auf eine Grafikstruktur benötigt, die bei Aktivierung des Gadgets im Austausch mit der alten Grafik gezeichnet werden soll.

***gg_GadgetText**
Zeiger auf eine IntuiText-Struktur, die den Text des Gadgets enthält oder Null, wenn auf einen Text verzichtet wird.

gg_MutualExclude
Die nächste Funktion wäre recht interessant, würde sie von Intuition unterstützt werden. Es handlet sich hierbei um die Exclude-Funktion, auf die wir an dieser Stelle nicht näher eingehen wollen, da sie für die Gadgets nicht implementiert ist. Man sollte hier immer eine Null einsetzen!

gg_SpecialInfo
Zeiger auf eine SpecialInfo-Struktur, die je nach Typ das Gadgets besondere Informationen enthalten kann. Bei Boolean-Gadgets kann man hier einen Zeiger auf eine BoolInfo-Struktur einsetzen, dann muß jedoch das BOOLEXTEND-Flag des Activation Eintrags gesetzt sein. In unserem Beispiel lassen wir die BoolInfo-Struktur weg.

gg_GadgetID
Identifikationsnummer des Gadgets, die wir bei der Nachrichtenauswertung abfragen können um die Meldung einem Gadget zuzuordnen. Es ist natürlich auch möglich, mehreren Gadgets dieselbe Nummer zuzuweisen. Dies kann bei einigen Situationen recht praktisch sein.

gg_UserData
Zeiger auf eigene Daten, die vom Benutzer angelegt und verwaltet werden können.

5.4.2 Integer- und String-Gadgets

Die Integer- bzw. String-Gadgets dienen dazu, Zahlen bzw. Zeichenketten einzugeben. Da die Grundstruktur der Gadgets identisch ist, können wir durch einige wenige Veränderungen der beschriebenen Struktur, aus dem Boolean-Gadget ein String-Gadget machen. Zunächst ändern wir den Typ des Gadgets. Wir setzen in diesen Eintrag eine 4 ein um den STRGADGET-Typ einzustellen. Dann benötigen wir einen Zeiger auf eine String-Info-Struktur. Das wäre eigentlich schon alles. Man könnte allerdings noch die Größe und einige Flags nach Belieben umstellen. Nun aber zur String-Info-Struktur. Sie hat folgenden Aufbau:

StringInfo-Struktur:

```
00    dc.l   *si_Buffer          ; Zeiger auf Text-Puffer
04    dc.l   *si_UndoBuffer      ; Zeiger auf Undo-Puffer
08    dc.w    si_BufferPos       ; Position im Puffer
10    dc.w    si_MaxChars        ; maximale Anzahl Zeichen
12    dc.w    si_DispPos         ; Pos. des ersten Zeichens
14    dc.w    si_UndoPos         ; Position im Undo-Puffer
16    dc.w    si_NumChars        ; Anzahl Zeichen im Puffer
18    dc.w    si_DispCount       ; Ausgabeposition
20    dc.w    si_CLeft           ; relative X-Position zum Win
22    dc.w    si_CTop            ; relative Y-Position zum Win
24    dc.l   *si_LayerPtr        ; Zeiger auf Layer
28    dc.l    si_LongInt         ; Zahlernwert der Eingabe
32    dc.l   *si_AltKeyMap       ; eigene Tastaturtabelle
36            si_SIZEOF
```

***si_Buffer**
Zeiger auf einen Speicherbereich, in welchem die eingegebenen Zeichen stehen. Der Speicher kann schon eine Zeichenkette enthalten, die direkt angezeigt werden soll.

***si_UndoBuffer**
Jede Eingabe kann durch die Tastenkombination Amiga-Q rückgängig gemacht werden. Hierzu muß die letzte Zeichenkette zwischengespeichert werden. Die Adresse des Zwischenspeichers muß in dem Eintrag UndoBuffer eingesetzt werden.

si_BufferPos
Cursorposition in der Zeichenkette.

si_MaxChars
Maximale Anzahl der Zeichen, die eingegeben werden können. Dieser Eintrag sollte mit der Größe des Speichers für die Zeichenkette abgestimmt sein. Wurde die maximale Anzahl erreicht, so wird dies durch einen DisplayBeep angezeigt.

si_DispPos
DisplayPosition gibt die Position in der Zeichenkette, ab welcher die Zeichen im Container (Eingabefeld) sichtbar sein sollen, an.

si_UndoPos

Position des Cursors im Undo-Puffer.

si_NumChars
Anzahl der Zeichen, die sich im Puffer befinden.

si_DispCount
Maximale Anzahl der Zeichen die angezeigt werden können.

Dem Eintrag DispCount folgen vier Parameter, deren Verwaltung von Intuition übernommen wird. Natürlich kann man die teilweise recht interessanten Werte auslesen. Deshalb sollen sie erklärt werden.

si_CLeft, si_CTop
Durch diese Werte ist die relative Position des Gadgets in bezug auf die linke obere Fensterecke angegeben.

***si_LayerPtr**
Zeiger auf die Layer-Struktur.

si_LongInt
Bei einem Integer-Gadget kann man an dieser Stelle den Wert des eingegebenen ASCII-Strings auslesen.

***si_AltKeyMap**
Zeiger auf die Tastatur-Tabelle, die für die Eingabe benutzt werden soll. Den Standardwert bekommen wir auch hier durch eine Null.

Auf folgende Eingaben reagieren String-/Integer-Gadgets:	
Return/Enter:	Eingabe beenden
rechts/links:	Cursor durch den Text bewegen
Shift rechts/links:	Cursor an den Anfang/Ende setzen
Delete/Backspace:	Zeichen löschen (aktuell/links)
rechte Amiga + X:	löscht bestehende Eingabe
rechte Amiga + Q:	setzt den Inhalt des Undo-Buffers

5.4.3 Proportional-Gadget

Kommen wir nun zum letzten Gadget-Typ, dem Proportional-Gadget. Sicher kennen sie diesen Typ schon, ein Beispiel für die Anwendung eines solchen Typs ist die Einstellung der Farben im Preferences-Programm. Dort kann man die Farbwerte mittels eines solchen Schiebeschalters festlegen. Oder auch in Dateiauswahl-Requestern ist diese Art oft anzutreffen. Der besondere Witz dieses Gadgets liegt in der proportionalen Größenveränderung des Schiebers. So kann man z.B. verfolgen, daß bei einigen Filerequestern während des Ladevorgangs des Inhaltsverzeichnises die Schiebergröße mit der Anzahl der gelesenen Dateien abnimmt.

Das Proportional-Gadget braucht eine gewöhnliche Gadget-Struktur. Wie bei den String-Gadgets tragen wir den Typ des Gadgets und einen Zeiger auf eine Special-Info-Struktur ein. Betrachten wir uns nun die PropInfo-Struktur.

PropInfo-Struktur:

```
00      dc.w    pi_Flags            ; Flags
02      dc.w    pi_HorizPot         ; X-Position des Schalters
04      dc.w    pi_VertPot          ; Y-Position des Schalters
06      dc.w    pi_HorizBody        ; X-Größe des Schalters
08      dc.w    pi_VertBody         ; Y-Größe des Schalters
10      dc.w    pi_CWidth           ; Breite des Gadgets
12      dc.w    pi_CHeight          ; Höhe des Gadgets
14      dc.w    pi_HPotRes          ; Schrittweite in X-Richtung
16      dc.w    pi_VPotRes          ; Schrittweite in Y-Richtung
18      dc.w    pi_LeftBorder       ; Position des Rahmens (X)
20      dc.w    pi_TopBorder        ; Position des Rahmens (Y)
22              pi_SIZEOF
```

pi_Flags
Einstellungen, die das Gadget betreffen:

Prop-Flags	Wert	Bedeutung
FREEHORIZ	$0002	horizontale Bewegungen erlauben
FREEVERT	$0004	vertikale Bewegungen erlauben
AUTOKNOB	$0001	Schieber wird von Intuition erstellt
PROPBORDERLESS	$0008	keinen Rahmen um das Gadget zeichnen
KNOBHIT	$0100	Schieber ist betätigt worden

FREEHORIZ Dieses Flag erlaubt die horizontale Bewegung.

FREEVERT Dieses Flag erlaubt die vertikale Bewegung.

AUTOKNOB Wenn kein eigener Schieber angegeben wird, kann man mit dem Flag AUTOKNOB Intuition veranlassen, einen Schieber zu erstellen.

PROPBORDERLESS Soll das Proportional-Gadget ohne Rand dargestellt werden muß dieses Flag gesetzt werden.

KNOBHIT KNOBHIT ist ein Informations-Flag. Wurde der Schieber mit dem Mauszeiger aktiviert, so ist es gesetzt.

pi_HorizPot, pi_VertPot
Die folgenden Einträge sind für die vertikale und horizontale Position des Schiebers verantwortlich. Dabei gilt der Wert 0 für links bzw. oben und der Wert $FFFF (-1) für rechts bzw. unten.

pi_HorizBody, pi_VertBody
Die Einträge HorizBody und VertBody geben die Schrittgröße an, mit der sich der Schieber bewegen soll. Auch diese Einstellung kann zwischen 0 und $FFFF liegen. Wenn man z.B. 20 Schritte einstellen will, muß man den Wert $FFFF/20 einsetzen.

Die folgenden sechs Werte werden von Intuition selbst gesetzt und können wärend des Programms abgefragt werden.

pi_CWidth, pi_CHeight
Breite und Höhe der Hit-Box.

pi_HPotRes, pi_VPotRes
Erhöhung der vertikalen und horizontalen Werte.

pi_LeftBorder, pi_TopBorder
Position der Umrandung des Proportional-Gadgets.

Für diesen Gadget-Typ stellt uns Intuition zwei wichtige Funktionen zur Verfügung.

ModifyProp	=	**-156 (Intuition-Library)**

***Gadget**	**a0**	<	Zeiger auf die PropGadget-Struktur, die verändert werden soll
***Window**	**a1**	<	Zeiger auf das Fenster, in dem das Gadget liegt oder Null
***Requester**	**a2**	<	Zeiger auf den Requester, in dem das Gadget liegt oder Null
Flags	**d0**	<	Neue Flags, die eingestellt werden sollen
HPot	**d1**	<	horizontale Position des Schiebers
VPot	**d2**	<	vertikale Position des Schiebers
HBody	**d3**	<	Breite des Schiebers
VBody	**d4**	<	Höhe des Schiebers
Erklärung			Durch ModifyProp kann man die Einstellungen eines Prop-Gadgets nachträglich ändern. Wurden die neuen Werte initialisiert, werden alle Gadgets ab dem angegebenen PropGadget neu gezeichnet.

NewModifyProp	=	**-468 (Intuition-Library)**

***Gadget**	**a0**	<	Zeiger auf die PropGadget-Struktur, die verändert werden soll.
***Window**	**a1**	<	Zeiger auf das Fenster, in dem das Gadget liegt oder Null.
***Requester**	**a2**	<	Zeiger auf den Requester, in dem das Gadget liegt oder Null.
Flags	**d0**	<	Neue Flags, die eingestellt werden sollen.
HPot	**d1**	<	Horizontale Position des Schiebers.
VPot	**d2**	<	Vertikale Position des Schiebers.
HBody	**d3**	<	Breite des Schiebers.
VBody	**d4**	<	Höhe des Schiebers.
Num	**d5**	<	Anzahl der weiteren Gadgets, die nach dem angegebenen Prop-Gadget aufgefrischt werden sollen.

Erklärung

Durch NewModifyProp kann man, wie schon mit der Funktion ModifyProp, die Einstellungen eines ProportionalGadget verändern. Der Unterschied beider Funktionen liegt darin, daß man bei der NewModifyProp-Funktion die Anzahl der Gadgets, die aufgefrischt werden sollen, angeben kann.

5.4.4 Gadget-Abfrage

Nachdem wir die Unterschiede der Gadgets kennengelernt haben, sollten wir uns jetzt noch um ihre Abfrage kümmern. Dazu können wir dieselbe Nachrichtenbehandlung, die bei den Fenstern benutzt wird, verwenden. Bisher haben wir immer darauf gewartet, daß das CloseGadget des Fensters betätigt worden ist. Nun müssen wir auch prüfen, ob ein Gadget aktiviert worden ist. Dazu dienen die beiden IDCMP-Flags GADGETDOWN und GADGETUP. Ist nun ein Gadget aktiviert worden, müssen wir auch die Identifikationsnummer des Gadgets erhalten. Sie wird jedoch nicht in der IntuiMessage-Struktur eingetragen. Anstelle dessen finden wir im Eintrag IAddress einen Zeiger auf die Gadget-Struktur des Gadgets, welches ausgelöst wurde. Nun können wir ganz einfach aus der angegebenen Gadget-Struktur den ID-Eintrag auslesen.

```
        ...

MessageBranch:
        move.l  Message,a0      ; Class-Eintrag auslesen
        move.l  20(a0),Class
        move.l  28(a0),a0       ; IAddress nach a0
        move.w  38(a0),GadID    ; Gadget-Identifikationsnr.

        move.l  Message,a1      ; Nachricht quittieren
        jsr     ReplyMsg(a6)

        cmp.l   #$40,Class      ; Gadget betätigt ?
        beq     GadgetBranch    ; Ja, dann untersuchen

        bra     MessageLoop     ; Wiederholen

        ...

GadgetBranch:

        cmp.w   #1,GadID        ; Gadget mit der ID-Nummer 1,
        beq     Exit            ; dann Programm beenden

        bra     MessageLoop     ; Neue Nachricht holen

        ...
```

```
Class:          dc.l    0               ; IDCMP-Flag der Nachricht
GadID:          dc.l    0               ; Gadget-Identifikationsnummer
```

Bild 5.8: Abfrage von Gadgets

Haben wir nun die Gadgetnummer isoliert, können wir zwischen den einzelnen Gadgets unterscheiden und richtig reagieren. Wenn man viele Gadgets verwalten will, ist es ratsam, die Adressen der Routinen in einer Tabelle abzulegen. Wählt man die Gadget-ID Nummern sinnvoll, kann man die Position in der Tabelle, an der die Adresse der Routine steht, leicht ausrechnen und zu ihr verzweigen. Diese Methode wird auch beim Demoprogramm für die Intuition-Library gewählt.

```
        ....

GadgetBranch:
        cmp.w   #6,GadID        ; kontrollieren ob GadgetID-
        bgt     MessageLoop     ; Nummer im erlaubten Bereich liegt

        lea     Tabelle,a0      ; Anfang der Tabelle nach a0 laden

        moveq   #0,d0           ; d0 löschen
        move.w  GadID,d0        ; Gadgetnummer nach d0
        sub.l   #1,d0           ; GadID-Nummer erniedrigen!

        lsl.l   #2,d0           ; mit vier multiplizieren (lsl.l #2,d0
                                ; geht schneller als mulu)

        move.l  (a0,d0),a0      ; Adresse der betreffenden Routine aus
                                ; der Tabelle lesen
        jmp     (a0)            ; und zu ihr verzweigen.

        ....

Tabelle:
        dc.l    Exit            ; Adr. der Routine für Gad #1
        dc.l    LoadIFF         ; Adr. der Routine für Gad #2
        dc.l    SaveIFF         ; Adr. der Routine für Gad #3
        dc.l    LoadRAW         ; Adr. der Routine für Gad #4
        dc.l    SaveRAW         ; Adr. der Routine für Gad #5
        dc.l    Info            ; Adr. der Routine für Gad #6
```


Bild 5.9: Reaktion auf Gadget-Klicks

Abgesehen von denen, die wir bereits besprochen haben, gibt es nur noch wenige Funktionen, die für Gadgets verantwortlich sind. Dazu gehören die folgenden, die in alphabetischer Reihenfolge geordnet sind. Bei manchen Funktionen ist ein Pointer auf eine Window- und eine Requester-Struktur verlangt. Es muß jedoch nur eine Adresse übergeben werden. Die andere muß auf Null gesetzt werden.

ActivateGadget	=	-462 (Intuition-Library)

*Gadget	a0	<	Zeiger auf die Struktur des Gadgets, welches aktiviert werden soll.
*Window	a1	<	Zeiger auf die Struktur des Fensters, in dem das Gadget eingebunden ist oder Null.
*Requester	a2	<	Zeiger auf die Struktur des Requesters, in dem das Gadget eingebunden ist, oder Null.

Erklärung

Durch ActivateGadget kann das angegebene Gadget aktiviert werden.

AddGadget	=	-42 (Intuition-Library)

*Window	a0	<	Zeiger auf die Window-Struktur, in der das neue Gadget eingebunden werden soll.
*Gadget	a1	<	Zeiger auf die Struktur des Gadgets, welches in ein bestimmtes Fenster aufgenommen werden soll.
Position	d0	<	Position, an der das Gadget in die Liste eingebunden werden soll (-1 = Ende).
RealPos	d0	>	Listenposition, an der das Gadget eingefügt worden ist.

Erklärung

Durch die AddGadget-Funktion kann man ein Gadget nachträglich in ein Fenster einbinden. Nachdem die Funktion ausgeführt wurde, ist das Gadget zwar in die Liste aufgenommen worden, jedoch ist es noch nicht sichtbar. Hierzu bietet sich die Funktion RefreshGadgets oder RefreshGList an.

AddGList	=	-438 (Intuition-Library)

*Window	a0	<	Zeiger auf die Window-Struktur, in die die Gadgets eingebunden werden sollen oder Null.
*Gadget	a1	<	Zeiger auf das erste Gadget der Liste, die eingebunden werden soll.
*Requester	a2	<	Zeiger auf den Requester, in den die Gadgets eingebunden werden sollen oder Null.
Position	d0	<	Position, an der die Gadgets in die Liste eingebunden werden sollen (-1 = Ende).

NumGad	d1	<	Anzahl der Gadgets die eingefügt werden sollen.
RealPos	d0	>	Listenposition, an der das Gadget eingefügt worden ist.
Erklärung			Diese Funktion fügt die angegebene Anzahl Gadgets aus der übergebenen Gadgetliste an der angegebenen Position des Requesters oder Fensters ein.

OffGadget	**= -174 (Intuition-Library)**

***Gadget**	a0	<	Zeiger auf die Struktur des Gadget, welches ausgeschaltet werden soll.
***Window**	a1	<	Zeiger auf das Fenster, in dem das Gadget eingebunden ist oder Null.
***Requester**	a2	<	Zeiger auf den Requester, in dem das Gadget eingebunden ist oder Null.
Erklärung			Die Funktion OffGadget schaltet ein Gadget aus. Das heißt, die HITBOX wird schattiert dargestellt und das Gadget kann nichtmehr betätigt werden. (Siehe auch OnGadget.)

OnGadget	**= -186 (Intuition-Library)**

***Gadget**	a0	<	Zeiger auf die Struktur des Gadgets, welches eingeschaltet werden soll.
***Window**	a1	<	Zeiger auf das Fenster, in dem das Gadget eingebunden ist oder Null.
***Requester**	a2	<	Zeiger auf den Requester, in dem das Gadget eingebunden ist oder Null.
Erklärung			Die Funktion OnGadget schaltet ein Gadget ein. Danach kann es wieder normal benutzt werden (siehe auch OffGadget).

RefreshGadgets	**= -222 (Intuition-Library)**

***Gadget**	a0	<	Zeiger auf die Struktur des Gadgets, welches neu gezeichnet werden soll.
***Window**	a1	<	Zeiger auf das Fenster, in dem das Gadget eingebunden ist oder Null.
***Requester**	a2	<	Zeiger auf den Requester, in dem das Gadget eingebunden ist oder Null.
Erklärung			Diese Funktion frischt das angegebene Gadget und alle folgenden auf (siehe auch RefreshGList).

RefreshGList	=	**-432 (Intuition-Library)**

***Gadget**	**a0**	<	Zeiger auf die Struktur des ersten Gadgets der Gadgetliste, die aufgefrischt werden soll.
***Window**	**a1**	<	Zeiger auf das Fenster, in dem das Gadget eingebunden ist, oder Null.
***Requester**	**a2**	<	Zeiger auf den Requester, in dem das Gadget eingebunden ist oder Null.
NumGad	**d0**	<	Anzahl der Gadgets, die nach dem angegebenen neu gezeichnet werden sollen.

Erklärung

Das angegebene Gadget und eine bestimmte Anzahl folgender werden neu gezeichnet (siehe auch RefreshGadget).

RemoveGadget	=	**-228 (Intuition-Library)**

***Window**	**a0**	<	Zeiger auf das Fenster, in dem das Gadget eingebunden ist.
***Gadget**	**a1**	<	Zeiger auf die Struktur des Gadgets, welches entfernt werden soll.

Erklärung

Das angegebene Gadget wird aus der Liste der Gadgets des angegebenen Fensters entfernt. Dabei werden die dazugehörigen Grafikstrukturen, die auf dem Fenster gezeichnet worden sind, nicht gelöscht.

RemoveGList	=	**-444 (Intuition-Library)**

***Window**	**a0**	<	Zeiger auf das Fenster in dem das Gadget eingebunden ist.
***Gadget**	**a1**	<	Zeiger auf die Struktur des ersten Gadgets der Liste, welche entfernt werden soll.
***NumGad**	**d0**	<	Anzahl der Gadgets, die entfernt werden sollen.

Erklärung

Das angegebene Gadget und die angegebene Anzahl der folgenden Gadgets werden entfernt. Dabei werden die dazugehörigen Grafikstrukturen, die auf dem Fenster gezeichnet worden sind, nicht gelöscht.

5.5 Menüs

Die Menüs sind neben den Gadgets die beliebteste Möglichkeit, eine strukturierte Programmoberfläche zu realisieren. Doch schon beim Einbinden der Menüs zeigt sich ein grundlegender Unterschied gegenüber den Gadgets. Während es für User-Gadgets die Möglichkeit des Einbindens in die NewWindow-Struktur gibt, muß man die Menüs von "Hand" implementieren. Aber so schlimm ist das nun auch wieder nicht, da uns Intuition kraftvoll mit einer Funktion unter die Arme greift. Gemeint ist die SetMenuStrip-Funktion, die als Parameter einen Zeiger auf die Menü-Struktur bzw. Kette verlangt. Da die Menüs abhängig von einem Fenster sind, ist es nicht verwunderlich, daß wir als zweiten Parameter einen Zeiger auf unser Window übergeben müssen.

SetMenuStrip	=	**-264 (Intuition-Library)**

***Window**	**a0**	<	Zeiger auf eine Window-Struktur
***Menu**	**a1**	<	Zeiger auf die erste Menü-Struktur

Erklärung Die angegebene Menüliste wird in die Fenster-Struktur implementiert.

Natürlich kann man die Menü-Strukturen auch wieder entfernen. Dazu dient die Funktion ClearMenuStrip.

ClearMenuStrip	=	**-54 (Intuition-Library)**

***Window**	**a0**	<	Zeiger auf eine Window-Struktur

Erklärung Die Funktion ClearMenuStrip entfernt die Menüs, die mit SetMenuStrip für dieses Fenster angegeben wurden.

Im Programm sieht das Ganze dann so aus:

```
SetMenuStrip =      -264
ClearMenuStrip      =           -54

        ...
        move.l  IntBase,a6
        move.l  WindowHD,a0
        lea     Menu1,a1
        jsr     SetMenuStrip(a6)

        ...
```

```
        move.l  IntBase,a6
        move.l  WindowHD,a0
        jsr     ClearMenuStrip(a6)
        ...
```

Bild 5.10: Aufruf von SetMenuStrip und ClearMenuStrip

Um ein Menü zu erstellen, brauchen wir zwei Strukturen. Die Menu-Struktur, in der die Einträge der Menüleiste abgelegt werden, sowie die MenuItem-Struktur, welche die eigentlichen Auswahlmöglichkeiten darstellt. Als Basis dient die Menu-Struktur. Deshalb beginnen wir mit ihr.

Die Menu-Struktur:

```
00   dc.l   *mu_NextMenu      ; Zeiger auf nächstes Menü
04   dc.w    mu_LeftEdge      ; X-Position
06   dc.w    mu_TopEdge       ; Y-Position
08   dc.w    mu_Width         ; Breite des Menüs
10   dc.w    mu_Height        ; Höhe des Menüs
12   dc.w    mu_Flags         ; Flags
14   dc.l   *mu_MenuName      ; Zeiger auf Menüname
18   dc.l   *mu_FirstItem     ; Zeiger auf ersten Menupunkt
22   dc.w    mu_JazzX         ; private Variablen Intuition
24   dc.w    mu_JazzY         ; private Variablen Intuition
26   dc.w    mu_BeatX         ; private Variablen Intuition
28   dc.w    mu_BeatY         ; private Variablen Intuition
30           mu_SIZEOF
```

***mu_NextMenu**
Zeiger auf nächste Menü-Struktur. Der erste Wert zeigt schon, daß es sich um verkettete Strukturen handelt. Man kann hier einen Zeiger auf eine weitere Menu-Struktur "einhängen".

mu_LeftEdge, mu_TopEdge
Position des Menüpunktes in der Titelzeile. Dabei wird allerdings <u>nur</u> der Wert von LeftEdge berücksichtigt.

mu_Width, mu_Height
Breit und Höhe des Menüpunktes. Auch hier wird der zweite Wert (Height/Höhe) nicht berücksichtigt, da er immer auf die Höhe der Titelzeile festgelegt ist.

mu_Flags
In Flags kann man folgende Werte setzen:

Menü-Flags	Wert	Bedeutung
MENUENABLED	$0001	Menüpunkt ist nicht anwählbar
MIDDRAWN	$0100	Menüpunkt wird gerade angezeigt

MENUENABLED	Die Menüpunkte des Menüs können nicht ausgewählt werden.
MIDDRAWN	Die Tabelle der Menüpunkte ist im Augenblick "ausgeklappt" (wird von Intuition benutzt).

***MenuName**
Zeiger auf eine Zeichenkette, die in der Menüzeile ausgegeben werden soll (der Menüname).

***mu_FirstItem**
Zeiger auf die verwendeten Menüitems, deren Struktur gleich erklärt wird.

mu_JazzX, mu_JazzY, mu_BeatX, mu_BeatY
Abgeschlossen wird die Struktur durch vier Words, die von Intuition intern benutzt werden.

Da wir nicht nur die Menüzeilen, sondern auch Menüpunkte, einbinden wollen, müssen wir jetzt noch die 12 Parameter der MenüItem-Struktur untersuchen.

Die MenüItem-Struktur:

```
00    dc.l   *mi_NextItem         ; nächste MenuItem-Struktur
04    dc.w    mi_LeftEdge         ; X-Koordinate des Punktes
06    dc.w    mi_TopEdge          ; Y-Koordinate des Punktes
08    dc.w    mi_Width            ; Breite des Menüpunktes
10    dc.w    mi_Height           ; Höhe des Menüpunktes
12    dc.w    mi_Flags            ; Flags des Menüpunktes
14    dc.l    mi_MutualExclude    ; MutualExclude-Daten
18    dc.l    mi_ItemFill         ; Grafikdaten "normal"
22    dc.l    mi_SelectFill       ; Grafikdaten "aktiviert"
26    dc.b    mi_Command          ; Tastaturcode
27    dc.b    mi_KludgeFill00     ; Füllbyte
28    dc.l   *mi_SubItem          ; Zeiger auf Untermenü
32    dc.l    mi_NextSelect       ; Nächster ausgewählter Menüpunkt
36            mi_SIZEOF
```

***mi_NextItem**
Zeiger auf die nächste MenuItem-Strukur, die eingebunden werden soll oder eine Null, wenn keine weitere Struktur benutzt werden soll.

mi_LeftEdge, mi_TopEdge, mi_Width, mi_Height
Position und Größe des Menüpunktes.

mi_Flags
Der Eintrag Flags legt die Eigenschaften des Menüpunktes fest. Die Bedeutung der Flags soll die anschließende Tabelle klären.

Menuitem-Flag	Wert	Bedeutung
CHECKIT	$0001	Menüpunkt abhaken
ITEMTEXT	$0002	IntuiText-Strukturen werden verwendet
COMMSEQ	$0004	Menü kann durch Tasten angewählt werden
MENUTOGGLE	$0008	der Zustand wird umgedreht
ITEMENABLED	$0010	Menü ist wählbar
HIGHIMGE	$0000	bei Aktivierung neues Image anzeigen
HIGHCOMP	$0040	bei Aktivierung Bereich invertieren
HIGHBOX	$0080	bei Aktivierung Bereich umranden
HIGHNONE	$00C0	bei Aktivierung passiert nichts
HIGHITEM	$2000	Menüpunkt ist gerade aktiviert
CHECKED	$0100	Menüpunkt ist gewählt
ISDRAWN	$1000	Menüpunkt wird dargestellt
MENUTOGGLED	$4000	Menüpunkt wurde schon umgedreht

CHECKIT Der Menüpunkt wird bei Selektion abgehakt bzw. der Haken wird wieder entfernt (je nachdem, ob das MENUTOGGLE-Flag gesetzt ist). Die Grafikdaten des Hakens konnten wir in der NewWindow-Struktur bestimmen.

ITEMTEXT Die Einträge ItemFill und SelectFill sind nun Zeiger auf IntuiText-Strukturen. Sonst wurden Image-Strukturen erwartet.

COMMSEQ Der Menüpunkt kann durch eine Tastenkombination angewählt werden. Die Tastenkombination besteht aus der rechten Amiga-Taste und einer zweiten, in Command angegebenen Taste.

MENUTOGGLE Der Zustand des Menüpunktes (abgehakt oder nicht) wird bei jeder Selektion umgedreht. Dabei wird auch das Flag CHECKED verändert.

ITEMENABLED Der Menüpunkt und seine Unterpunkte können ausgewählt werden.

HIGHIMGE Ist das Menü aktiviert worden, wird das Image, auf den der Zeiger SelectFill zeigt, ausgegeben.

HIGHCOMP Wenn der Menüpunkt angewählt wurde, wird der angegebene Bereich des Menüpunktes invertiert dargestellt.

HIGHBOX Der Menüpunkt wird mit einem Rahmen umgeben, wenn er aktiviert worden ist.

HIGHNONE Es passiert nichts, wenn das Menü aktiviert wird.

HIGHITEM Das Falg HIGHITEM ist immer dann gesetzt, wenn der Menüpunkt mit dem Mauszeiger ausgewählt wurde.

CHECKED Ist der Haken an einem Menüpunkt gesetzt worden, ist das CHECKED-Flag aktiviert. Hierzu muß das Flag CHECKIT gesetzt sein !

ISDRAWN Durch ISDRAWN wird angezeigt, daß das Menü gerade dargestellt wird.

MENUTOGGLED Der Menüpunkt ist schon umgedreht worden.

mi_MutualExclude

Im Eintrag MutualExclude werden die Daten für diese Funktion gespeichert. Diese Funktion ist bei den Gadgets zwar vorge-

sehen, aber nicht realisiert worden. Bei den Menüs jedoch kann man von dieser praktischen Unterstützung Gebrauch machen. Die Funktionsweise ist wie folgt: Wird dieser Menüpunkt ausgewählt, kann dies Einfluß auf den Status (gewählt/nicht gewählt) der anderen Menüpunkte dieses Menütitels haben. Welcher Punkt ausgeschaltet werden soll, wird in diesem Langwort abgelegt. Dabei entspricht das erste Bit (Bit 0) dem ersten Menüpunkt und jedes folgende Bit jedem weiteren Menüpunkt. Soll der Zustand des Menüpunktes beeinflußt werden, muß das jeweilige Bit gesetzt werden.

mi_ItemFill
Zeiger auf eine Grafikstruktur, die im Menü gezeichnet werden soll. Normalerweise wird ein Zeiger auf eine Image-Struktur vermutet. Dies kann jedoch durch den Eintrag ITEMTEXT geändert werden.

mi_SelectFill
Zeiger auf Grafikdaten, die ausgegeben werden sollen, wenn der Menüpunkt aktiviert worden ist. Wie bei ItemFill ist auch hier der Typ der Struktur abhängig von dem Flag ITEMTEXT. Jedoch muß das Flag HIGHIMAGE gesetzt sein, damit bei aktivierten Menüitems die Grafiken bzw. Texte ausgetauscht werden.

mi_Command
Wie sie vielleicht schon wissen, kann man die Menüpunkte nicht nur mit dem Mauszeiger sondern auch mit der Tastatur anwählen. Hierzu muß angegeben werden, auf welche Taste, in Kombination mit der rechten Amiga-Taste, reagiert werden soll. Auch hier müssen wir die Tastenwahl der Menüpunkte im Flag-Eintrag der Struktur extra "erlauben". Hierzu dient das COMMSEQ-Flag.

mi_KludgeFill00
Da der Eintrag Command nur ein Byte lang ist und der PC an dieser Stelle einen ungeraden Wert enthalten würde, benötigt man diesen Füllbyte-Eintrag, um ihn auf Wort-Grenze "umzubiegen".

mi_SubItem
Zeiger auf weitere MenüItem-Strukturen, die als Unterpunkte verwendet werden sollen. Diese Verschachtelung ist jedoch nur einmal möglich, bei der nächsten Struktur wird dieser Eintrag nicht berücksichtigt.

mi_NextSelected
Das letzte Langwort enthält, falls mehrere Menüpunkte ausgewählt worden sind, die Nummer des nächsten gewählten Menüpunktes.

Nachdem wir uns durch soviele trockene Daten-Strukturen gearbeitet haben, kommen wir jetzt wieder zur Programmierung. Wie bei den Gadgets müssen wir auch bei der Menübearbeitung darauf achten, daß wir die Meldung im IDCMP-Parameter des Windows erlaubt haben. Sonst warten wir vergebens auf eine

Nachricht von Intuition. Wie wir nun die Nummer des Menüs, des Menüpunktes und eventuell auch des Menüunterpunktes aus der IntuiMessage-Struktur lesen, sehen wir uns in dem folgenden Programmabschnitt an.

```
        ...
MessageBranch:
        move.l   Message,a0       ; Auslesen der Class- und
        move.l   20(a0),Class     ; Code-Einträge
        move.w   24(a0),Code

        move.l   Message,a1       ; Nachricht bestätigen
        jsr      ReplyMsg(a6)

        cmp.l    #$100,Class      ; Class-Eintrag kontrollieren
        beq      MenuLoop         ; Menü ausgewält!

        ...                       ; hier können andere Abfragen
                                  ; eingesetzt werden
        bra      MessageLoop      ; zurück und auf nächste
                                  ; Nachricht warten!

        ...

MenuDecoder:                      ; Menüabfrage

        move.w   Code,d0          ; Code-Eintrag nach d0
        cmp.w    #-1,d0           ; Testen ob überhaupt ein
        beq      MessageLoop      ; Menüpunkt gewählt wurde

        and.l    #%11111,d0       ; Alle Bits bis auf die unteren 5
                                  ; ausblenden, um Menünummer zu erhalten

        moveq    #0,d1            ; d1 löschen
        move.w   Code,d1          ; Code-Eintrag nach d1
        lsr.l    #5,d1            ; d1 um 5 Bits nach rechts
        move.w   d1,d2            ; neuen Wert nach d2 kopieren
        and.l    #%111111,d1      ; Alle Bits bis auf die unteren 6
                                  ; ausblenden, um Menüpunktnummer zu
                                  ; erhalten

        lsr.l    #6,d2            ; Wert abermals nach rechts schieben, um
                                  ; Menüunterpunktnummer zu bekommen

        ; Menünummer             = d0
        ; Menüpunktnummer        = d1
        ; Menüunterpunktnummer = d2

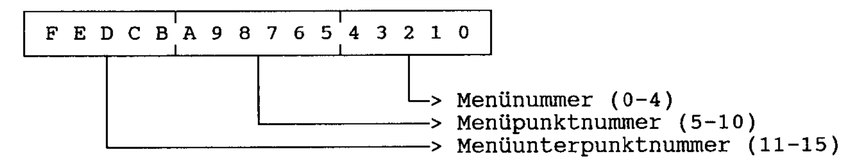
        ; Jetzt müssen lediglich die Datenregister 1-3 üperprüft werden.

        ...
```

Bild 5.11: Abfrage und Auswertung einer Menü-Auswahl

Zunächst wird in der Message-Branch-Routine kontrolliert, ob die angekommene Nachricht für die Menüauswahl gilt. Dann wird zur Unterroutine MenüDecoder verzweigt, die die Auswertung des Eintrags "Code" vornimmt, in welchem der ausgewählte Menüpunkt eingetragen ist. Der Eintrag unterteilt sich in drei Gruppen:

Code-Eintrag:



Bevor wir uns an die Entschlüsselung machen können, müssen wir kontrollieren, ob überhaupt ein Menüpunkt angewählt worden ist. Es kann ja durchaus sein, daß die rechte Maustaste nicht über einem Menüpunkt, sondern irgendwo anders losgelassen wurde. Sollte das der Fall gewesen sein, sind alle Bits gesetzt. Das heißt, der Eintrag Code hat den Wert $FFFF (oder -1). Nachdem wir das kontrolliert haben, können wir uns an die Isolierung der Menünummer machen. Das erreichen wir, wenn wir alle Bits bis auf die ersten fünf ausblenden.

```
        and.l    #%11111,d0      ; Bits ausblenden
```

Man sollte eine Langwort-Operation durchführen, damit das obere Wort von d0 gelöscht wird.

Um die Nummer des ausgewählten Menüpunktes zu bekommen, müssen wir die 16 Bits des Code-Wertes fünf Stellen nach rechts schieben. Bevor wir die überflüssigen Bits wieder ausblenden, retten wir den manipulierten Wert nach d2, um ihn später zur Auswertung des Menü-Unterpunktes zu benutzen.

```
        moveq    #0,d1           ; Datenregister löschen
        move.w   Code,d1         ; Code übertragen
        lsr.l    #5,d1           ; fünf Bits nach rechts
        move.w   d1,d2           ; und Wert für später retten.
        and.l    #%111111,d1     ; Nun die störenden Bits ausblenden
```

Die Nummer des Menü-Unterpunktes können wir jetzt mit einem Befehl festlegen. Wir schieben lediglich die 6 Bits für den Menüpunkt aus dem Datenregister "hinaus".

Jetzt, wo die einzelnen Werte isoliert in den Datenregistern d0-d2 enthalten sind, können wir nach Belieben die Register testen. Um die Auswahl der richtigen Routine etwas einfacher zu gestalten, kann man, wie schon bei den Gadgets, ihre Adressen in Tabellen ablegen.

5.6 Requester

Requester bieten dem Programmierer die Möglichkeit, Nachrichten an den Benutzer weiterzugeben und zusätzlich noch Eingaben auszuwerten.

5.6.1 Alerts

Die erste Möglichkeit, die wir uns ansehen wollen, sind die Alerts, die jeder Benutzer sofort mit den GURUS in Verbindung bringt. Die Funktion, mit der wir den Alert auf dem Bildschirm darstellen können heißt DisplayAlert.

DisplayAlert	**=**	**-90 (Intuition-Library)**

***String**	**a0**	<	Zeiger auf eine Struktur mit dem Text, der ausgegeben werden soll, und seiner Position.
AlertNumber	**d0**	<	Alertnummer.
Height	**d1**	<	Höhe des Alerts.
Response	**d0**	>	Gedrückte Maustaste.
Erklärung			Durch die Funktion DisplayAlert wird der angegebenen Text als Alert ausgegeben.

Der Wert, der im Datenregister 0 übergeben werden muß, gibt den Typ des Alerts an. Man unterscheidet dabei zwei verschiedene Formen:

```
DEADEND_ALERT     =     $80000000
RECOVERY_ALERT    =     $00000000
```

Unter einem DEADEND-Alert versteht man einen Alert, der in einen Reset endet, während der RECOVERY-Alert, wie der Name schon sagt, ins Programm zurückkehrt. Das heißt aber nicht, daß die DisplayAlert-Routine automatisch einen Reset nach einem DEADEND-Alert durchführt. Vielmehr wirkt sich die Alertnummer auf die Rückmeldung der gedrückten Maustaste aus: RECOVERY-Alerts geben eine -1 zurück, falls die linke Taste gedrückt wurde und eine 0 bei der rechten Taste. DEADEND-Alerts aber geben immer eine 0 zurück.

Die Daten, auf die wir in a0 einen Zeiger übergeben müssen, haben ein vordefiniertes Aussehen:

String:

```
00      dc.w    LeftEdge            ; X-Position des Textes
02      dc.b    TopEdge             ; Y-Position des Textes
03      ds.b    AlertText,nn        ; Zeichenkette
nn      dc.b    Flag                ; Flags
```

LeftEdge, TopEdge
Positionierung des Textes, dabei wird die Y-Koordinate durch einen Byte-Wert angegeben, da der Wertebereich (0 bis 255) ausreicht.

AlertText
Zeichenkette, die an der angegebenen Position ausgegeben werden soll. Wie alle benutzten Zeichenketten muß auch diese mit einem Null-Byte abgeschlossen sein.

Flag
Durch den Wert im Eintrag Flag kann man bestimmen, ob eine weitere Textzeile angebunden werden soll oder nicht.

0	=	keine weitere Textzeile	(ENDE)
ungleich 0	=	weiterer Text folgt	(NEXT)

Wurde das Flag mit einem Wert ungleich 0 initialisiert, werden direkt nach dem letzten Eintrag die nächsten Daten mit dem gleichen Aufbau erwartet.

Der Aufruf eines DisplayAlerts von Intuition sieht dann wie folgt aus:

```
* Programm 5.13: Ausgabe eines Alerts mit DisplayAlert

ExecBase      =       4
OldOpenLib    =       -408
CloseLib      =       -414
DisplayAlert  =       -90

Start:  move.l    ExecBase,a6      ; Intuition-Library öffnen
        lea       IntName,a1
        jsr       OldOpenLib(a6)
        move.l    d0,IntBase

Loop:   move.l    IntBase,a6
        lea       AlertData,a0     ; Alertdaten nach a0
        move.l    #0,d0            ; RECOVERY-ALERT
        move.l    #55,d1           ; Höhe der Box
        jsr       DisplayAlert(a6)       ; Alert ausgeben

        tst.l     d0               ; rechte Maustaste (RMT) ?
        beq       Loop             ; ja, dann Alert erneut
                                   ; ausgeben
        move.l    ExecBase,a6
```

```
        move.l   IntBase,a1
        jsr      CloseLib(a6)   ; Intuition-Library schließen
        rts

* Datenbereich

IntName:        dc.b    "intuition.library",0
                even
IntBase:        dc.l    0

AlertData:                              ; Alertdaten
                dc.w    240             ; X-Position
                dc.b    15              ; Y-Position
                dc.b    "!!!! ACHTUNG !!!!",0
                                        ; Zeichenkette
                dc.b    -1              ; (NEXT)

                dc.w    170
                dc.b    25
                dc.b    "Dies ist ein RECOVERY-Display-Alert",0
                dc.b    -1
                dc.w    180
                dc.b    45
                dc.b    "LMT - Ende          RMT - Schleife",0
                dc.b    0               ; (END)
                even
```

Programm 5.13: Ausgabe eines Alerts mit DisplayAlert

Abgesehen von der DisplayAlert-Funktion, gibt es noch eine weitere Möglichkeit, einen Alert auszugeben. Diese soll jedoch erst im Kapitel zur Exec-Library besprochen werden.

5.6.2 AutoRequest

Wenn man nicht direkt die "Holzhammer-Methode" (Alert) benutzen will, kann man auch auf eine einfühlsamere Weise seine Informationen ausgeben. Sicherlich ist Ihnen dieser Typ der Kommunikation bekannt. Man denke nur an "Please insert Volume ... in Drive..." oder "Volume ... has a Read/Write Error" oder die etwas härtere Form "You MUST replace ... !!!". Diese Art der Meldung heißt SystemRequester und kann auch vom Programmierer benutzt werden. Hierzu bietet ihm Intuition die Funktion AutoRequest an. Zwar ist ein AutoRequester nicht sehr flexibel, dafür aber einfach zu benutzen.

AutoRequest	=	**-348 (Intuition-Library)**

***Window** **a0** < Zeiger auf eine Window-Struktur oder eine Null

***BodyText**	a1	<	Zeiger auf eine IntuiText-Struktur, die die Meldung enthält
***PositivText**	a2	<	Zeiger auf eine IntuiText-Struktur für das positive Gadget
***NegativText**	a3	<	Zeiger auf eine IntuiText-Struktur für das negative Gadget
PositiveFlag	d0	<	Weitere IDCMP-Flags für die positive Abbruchbedingung
NegativeFlag	d1	<	Weitere IDCMP-Flags für die negative Abbruchbedingung
Width	d2	<	Breite des Requesters
Height	d3	<	Höhe des Requesters
Response	d0	>	Wurde das negative Gadget oder eins der angegebenen negativen Flags ausgelöst, erhalten wir eine Null zurück.
Erklärung			Durch die Funktion AutoRequest wird ein Requester mit der angegebenen Meldung erstellt. Der Requester kann durch eins der beiden Gadgets geschlossen werden oder durch eine Meldung, die mit dem übergebenen IDCMP-Flag übereinstimmt. Je nachdem, welche Möglichkeit (positiv/negativ) ausgewählt worden ist, erhält man einen Rückgabewert zurück.

Soll Intuition für den Requester ein eigenes Window öffnen, so trägt man in a0 an Stelle eines Zeiger auf ein Fenster eine Null ein.

Das Register a1 enthält einen Zeiger auf eine IntuiText-Struktur des Body-Textes, welcher z.B. eine Aufforderung oder Mitteilung sein könnte. In a2 und a3 stehen zwei weitere Zeiger auf IntuiText-Strukturen, welche die beiden Auswahlmöglichkeiten beschreiben sollen. Meist werden hier "Retry" und "Cancel" eingesetzt.

In d0 und d1 kann man die IDCMP-Flags (siehe Window-Struktur) übergeben, die für die Auslösung der positiven bzw. negativen Meldung verantwortlich sein sollen. Auch wenn die Werte mit Null initialisiert wurden, wird immer noch auf die Gadgets reagiert.

Die Einstellungen für Breite und Höhe des Requesters können in den Datenregistern d2 und d3 abgelegt werden.

Nachdem die AutoRequest-Funktion aufgerufen worden ist, kann man aus dem Wert in d0 erkennen, ob die negative oder positive Auswahlmöglichkeit benutzt wurde, um den Requester zu schließen. Ist d0 mit Null initialisiert, wurde das Neg-Gadget aktiviert (bzw. eines der angegebenen IDCMP-Flags für "Neg" betätigt). Sollte es Probleme beim Öffnen des Requesters geben, so wird anstelle dessen ein Alert mit den selben Parametern dargestellt.

Auch hier wollen wir uns anschließend den Aufruf durch ein Programm ansehen:

```
* Programm 5.14: Benutzung von AutoRequest

ExecBase            =           4
OldOpenLib          =           -408
CloseLib            =           -414
AutoRequest         =           -348

Start:  move.l   ExecBase,a6     ; Intuition-Library öffnen
        lea      IntName,a1
        jsr      OldOpenLib(a6)
        move.l   d0,IntBase

Loop:   move.l   IntBase,a6
        move.l   #0,a0           ; kein eigenes Fenster
        lea      BodyText,a1     ; Zeiger auf Body-Text
        lea      PosText,a2      ; Zeiger auf Positiv-Text
        lea      NegText,a3      ; Zeiger auf Negativ-Text
        move.l   #$10000,d0      ; positive IDCMP-Flags
        move.l   #$8000,d1       ; negative IDCMP-Flags
        move.l   #10,d2          ; Breite des Requesters
        move.l   #100,d3         ; Höhe des Requesters
        jsr      AutoRequest(a6)

        tst.l    d0              ; positiv oder Negativ ?
        beq      Loop            ; nein, dann Requester neu aufbauen
        move.l   ExecBase,a6
        move.l   IntBase,a1      ; Intuition-Libaray schließen
        jsr      CloseLib(a6)
        rts                      ; Programm beenden

* Datenbereich

IntName:      dc.b    "intuition.library",0
              even
IntBase:      dc.l    0

BodyText:                              ; IntuiText-Struktur für den
              dc.b    0,0,0            ; Body-Text
              even
              dc.w    30,60
              dc.l    0,BodyData,0
BodyData:
              dc.b    "Wollen Sie diesen AutoRequester beenden ???",0
              even

PosText:                               ; IntuiText-Struktur für den
              dc.b    0,0,0            ; Positiv-Text
              even
              dc.w    0,0
```

```
            dc.l    0,PosData,0
PosData:
            dc.b    "JA, weg mit dem Sch..",0
            even

NegText:                            ; IntuiText-Struktur für den
            dc.b    0,0,0           ; Negativ-Text
            even
            dc.w    0,0
            dc.l    0,NegData,0
NegData:
            dc.b    "Auf keinen Fall !",0
            even
```

Programm 5.14: Benutzung von AutoRequest

5.6.3 BuildSysRequest/FreeSysRequest

Wenn man sich nicht mit der Unterscheidung positiv/negativ zufrieden gegeben will, gibt es noch die Möglichkeit, die Abfrage der Ereignisse selbst in die Hand zu nehmen. Dazu brauchen wir die Funktion BuildSysRequest, welche eine Unterfunktion von AutoRequest ist. Ihre Parameter sind ähnlich den Parametern der AutoRequest-Funktion. Man erhält jedoch nach dem Aufruf einen Zeiger auf eine Fenster-Struktur. Mit Hilfe dieses Zeigers kann man die Abfrage des Requesters selbst bewerkstelligen.

BuildSysRequest = -360 (Intuition-Library)

***Window**	**a0**	<	Zeiger auf eine Window-Struktur oder Null
***BodyText**	**a1**	<	Zeiger auf eine IntuiText-Struktur (Body)
***PositivText**	**a2**	<	Zeiger auf eine IntuiText-Struktur (Pos)
***NegativText**	**a3**	<	Zeiger auf eine IntuiText-Struktur (Neg)
IDCMPFlags	**d0**	<	IDCMP-Flags des Fensters
Width	**d2**	<	Breite des Requesters
Height	**d3**	<	Höhe des Requesters
Window	**d0**	>	Adresse der Window-Struktur des geöffneten Fensters.

Erklärung

Durch die Funktion BuildSysRequest wird zwar ein Requester erstellt, muß die Abfrage der Ereignisse jedoch selbständig vom Programm übernommen werden. Dazu erhalten wir einen Zeiger auf eine Window-Struktur, in dem sich der Requester befindet.

Auch bei BuildSysRequest stellt Intuition eigene Gadgets mit einer positiven bzw. negativen Auswahlmöglichkeit zur Verfügung. Diese Gadgets muß man natürlich mit der eigenen Routine abfragen. Bei beiden Gadgets wurden die Flags BOOLGADGET, RELVERIFY, REQGADGET und TOGGLESELECT gesetzt.

Nachdem wir den Requester mittles der BuildSysRequest-Funktion aufgebaut haben, können wir über den zurückgelieferten Zeiger auf die Ereignisse eingehen. Dies geschieht, wie bereits im Abschnitt über die Windows besprochen wurde.

Zur Freigabe des Speichers steht Ihnen die FreeSysRequest-Funktion zur Verfügung. Sie erledigt alle nötigen Verwaltungsarbeiten, braucht dazu allerdings den Zeiger auf das Fenster, in dem der Requester gezeichnet worden ist. Diesen Zeiger haben wir beim Aufruf der BuildSysRequest-Funktion erhalten.

FreeSysRequest	**=**	**-372 (Intuition-Library)**

***Window**	**a0**	<	Zeiger auf Fenster in dem der Requester enthalten ist.
Erklärung			Gibt den durch den Requester belegten Speicher wieder frei.

Sollte ein Fehler beim Öffnen des Requesters eingetreten sein, so wird anstelle des Requesters ein Alert mit denselben Texten ausgegeben. Dann darf natürlich auch die Funktion FreeSysRequest nicht benutzt werden! (Das Demonstrationsprogramm befindet sich auf der Diskette.)

5.6.4 "Richtige" Requester

Als vierte und letzte Möglichkeit, eigene Requester zu erstellen, wollen wir uns die Request-Funktion ansehen. Mit Hilfe ihrer flexiblen Möglichkeiten kann man eine optimale Kommunikation zwischen Benutzer und Programm herstellen. Natürlich benötigt soviel Flexibilität auch einigen Aufwand, doch lohnt es sich mit Sicherheit, wenn man sein Programm verschönern will.

Auch für einen Requester gibt es eine Struktur, in die wir alle Werte eintragen müsssen.

Requester-Struktur:

```
000    dc.l    *rq_OlderRequest      ; Zeiger auf Requester
004    dc.w     rq_LeftEdge          ; X-Position
006    dc.w     rq_TopEdge           ; Y-Position
008    dc.w     rq_Width             ; Breite
010    dc.w     rq_Height            ; Höhe
012    dc.w     rq_RelLeft           ; Maus, relative X-Position
```

```
014    dc.w     rq_RelTop            ; Maus, relative Y-Position
016    dc.l    *rq_ReqGadget         ; Zeiger auf das erste Gadget
020    dc.l    *rq_ReqBorder         ; Zeiger auf einen Border
024    dc.l    *rq_ReqText           ; Zeiger auf einen Text
028    dc.w     rq_Flags             ; Flags
030    dc.b     rq_BackFill          ; Farb.Nr. des Hintergrundes
031    dc.b     rq_KludgeFill00      ; Füllbyte
032    dc.l    *rq_ReqLayer          ; Zeiger auf Layer-Struktur
036    ds.b     rq_ReqPad1,32        ; Reservierter Bereich
068    dc.l    *rq_ImageBMap         ; Zeiger auf eigenen BitMap
072    dc.l    *rq_RWindow           ; Zeiger auf das Req.-Fenster
076    ds.b     rq_ReqPad2,36        ; Reservierter Bereich
112             rq_SIZEOF
```

***rq_OlderRequest**
Zeiger auf eine andere Requester-Struktur (wird von Intuition benutzt).

rq_LeftEdge, rq_TopEdge, rq_Width, rq_Height
Position und Größenangabe des Requesters.

rq_RelLeft, rq_RelTop
Relative Positionierung des Requesters zur aktuellen Mauszeigerposition. Das heißt, der Requester wird nicht an den Koordinaten rq_LeftEdge, rq_TopEdge ausgegeben, sondern die Position errechnet sich aus der Mauszeigerposition und den hier angegebenen Offsetwerten. Um diesen Effekt einzuschalten, muß das Flag POINTREL gesetzt sein.

***rq_ReqGadget**
Zeiger auf das erste Gadget einer Gadgetliste, die in den Requester eingebunden werden soll. **ACHTUNG:** Mindestens eines der Gadgets sollte mit dem Activation-Flag ENDGADGET ausgerüstet sein. Sonst kann der Requester nur durch die Funktion EndRequest geschlossen werden.

***rq_ReqBorder**
Zeiger auf eine initialisierte Border-Struktur, die im Requester gezeichnet werden soll.

***rq_ReqText**
Zeiger auf eine initialisierte IntuiText-Struktur, die im Requester ausgegeben werden soll.

rq_Flags
Wie bei der relativen Positionierung gibt es auch hier wieder die Möglichkeit, mit Flags das Aussehen oder auch die Verwendung von Einträgen festzulegen.

Requester-Flag	Wert	Bedeutung
POINTREL	$0001	Position ist relativ zur Maus
PREDRAWN	$0002	Eigene BitMap einbinden
NOISYREQ	$0004	
REQOFFWINDOW	$1000	Requester außerhalb des Fensters
REQACTIVE	$2000	Requester aktiviert

SYSREQUEST	$4000	Requester ist ein System-Requester
DEFERREFRESH	$8000	Refreshmodus wird gestoppt

POINTREL	Requester relativ zur aktuellen Mauszeigerposition anzeigen. Dabei wird auf die Offsetwerte rq_RelLeft und rq_RelTop zurückgegriffen.
PREDRAWN	Anstelle der angegebenen IntuiText-, Border- und der in den Gadget-Strukturen angegebene Grafik-Strukturen wird die angegebene BitMap verwendet.
REQOFFWINDOW	Der Requester befindet sich außerhalb des angegebenen Fensters.
REQACTIVE	Der Requester ist im Augenblick aktiviert.
SYSREQUEST	Bei dem Requester handelt es sich um einen System-Requester (AutoRequest z.B.).
DEFERREFRESH	Keine Erneuerung.

rq_BackFill
Farbtabellennummer der Farbe, in der der Hintergrund des Requesters eingefärbt werden soll.

***rq_ReqLayer**
Zeiger auf die Layer-Struktur, die für den Requester zuständig ist.

rq_ReqPad1
Das 32 Byte große Array ReqPad1 wird von Intuition benutzt und hat interne Bedeutung.

***rq_ImageBMap**
Zeiger auf eine initialisierte BitMap-Struktur, die gezeichnet werden soll.

***rq_RWindow**
Zeiger auf das Fenster, in dem der Requester erstellt wurde.

rq_ReqPad2
Wie schon der Eintrag ReqPad1, sind auch diese Daten nur für Intuition angelegt.

Bevor wir den Requester aufrufen, erläutern wir noch die Funktion namens InitRequest. Durch sie wird die gesamte Requester-Struktur, auf die man einen Zeiger übergeben hat, automatisch mit Nullen gefüllt. Nun müßte man nachträglich vom Programm aus die Parameter in die Struktur schreiben. Diese Arbeit können wir uns sparen. Wir müssen nur darauf achten, daß alle Einträge, die keinen von uns bestimmten Wert enthalten, mit Null initialisiert sind. Dann können wir auf die InitRequest-Funktion verzichten und die Struktur direkt als Daten anlegen.

Trotzdem wollen wir Sie kurz aufführen.

InitRequest	=	-138 (Intuition-Library)

***Requester**	**a0**	<	Zeiger auf die Requester-Struktur, die gelöscht werden soll.
Erklärung			Durch die Funktion InitRequest werden alle Einträge der angegebenen Request-Struktur auf Null gesetzt. Nach dem Aufruf müssen dann alle Werte in die Struktur eingesetzt werden.

Um den Requester zu erstellen, benötigen wir die Request-Funktion.

Request	=	-240 (Intuition-Library)

***Requester**	**a0**	<	Zeiger auf eine Requester-Struktur
***Window**	**a1**	<	Zeiger auf eine Window-Struktur, in der der Requester erstellt und über dessen Message-Port die Kommunikation laufen soll.
Success	**d0**	>	Ist ein Fehler aufgetreten, erhält man eine Null zurück.
Erkärung			Durch die Request-Funktion wird ein Requester mit der angegebenen Struktur im angegebenen Fenster erstellt.

Wie man sieht, wird ein Zeiger auf das Fenster, in dem der Requester erstellt werden soll, benötigt. Über dessen MessagePort läuft dann auch die Kommunikation mit unserem Requester ab. Dazu sollte man sich jedoch eine spezielle Nachrichtenroutine für Requester schreiben.

Um den Requester wieder zu schließen, kennen wir bisher nur eine Möglichkeit: Ein Gadget mit dem ENDGADGET-Flag muß durch den Benutzer betätigt werden. Für den Programmierer bietet sich die Funktion EndRequest an.

EndRequest	=	-120 (Intuition-Library)

***Requester**	**a0**	<	Zeiger auf eine Requester-Struktur
***Window**	**a1**	<	Zeiger auf die Window-Struktur, in dem der Requester erstellt wurde.
Erklärung			Durch die Funktion EndRequest wird der Requester im angegebenen Fenster gelöscht. Die Nachrichten, die über den "Fensterkanal" ankommen, beziehen sich dann wieder auf das Fenster.

Mit dieser Funktion kann auch der Programmierer den Requester nach Belieben schließen.

5.6.5 DM-Requester

Durch die Request-Funktion kann man zwar einen Requester aufrufen, wobei jedoch der Zeitpunkt, an dem er gezeichnet wird, festgelegt ist. Eine flexiblere Art und Weise, einen Requester zu benutzen, bietet die SetDMRequest-Funktion. Durch sie wird der Requester an ein bestimmtes Fenster gebunden und immer dann aufgerufen, wenn ein DoubleClick mit der rechten Maustaste (Menütaste) ausgeführt worden ist.
Als DoubleClick bezeichnet man das zweimalige Betätigen einer Maustaste in dem von Preferences angegebenen Zeitraum. Deshalb heißt diese Funktion auch SetDoubleMenuRequest.

Ruft nun der Benutzer den installierten Requester auf, so sendet Intuition an unseren MessagePort eine Nachricht. Dies geschieht natürlich nur dann, wenn wir die Nachrichten im Eintrag IDCMP-Flags des Fensters "eingeschaltet" haben.

Gehen wir davon aus, daß dies erfolgt ist, so erhalten wir zunächst die Meldung REQVERIFY. Sie soll uns mitteilen, daß ein Requester von Intuition erstellt werden soll. Der Requester wird jedoch erst gebildet, wenn wir die Nachricht mittels ReplyMsg bestätigt haben. Danach bekommen wir die Meldung REQSE, was soviel heißt wie: Requester erstellt. Spätestens bei diesem Flag sollten wir in die gesonderte Nachrichtenroutine für Requester verzweigen!

Nun beziehen sich alle Nachrichten, die wir durch den MessagePort unseres Fensters erhalten, auf den Requester. Wurde das ENDGADGET betätigt, beendet Intuition den Requester und sendet uns die Nachricht REQCLEAR. Jetzt können wir mit der "normalen" Nachrichtenüberwachung weitermachen.

Jetzt aber endlich zu der SetDMRequest-Funktion!

SetDMRequest	=	**-258 (Intuition-Library)**

***Window**	**a0**	<	Zeiger auf das Fenster, in dem der Requester erscheinen und über dessen MessagePort nachher die Kommunikation ablaufen soll.
***Requester**	**a1**	<	Zeiger auf eine initialisierte Requester-Struktur.
Success	**d0**	>	Falls schon ein anderer Requester als DMRequester definiert ist, kommt in d0 eine 0 zurück, ansonsten eine -1.

Erklärung

Die Funktion SetDMRequest setzt einen Double-Menu-Requester, der durch einen

DoubleClick der rechten Maustaste ausgelöst werden kann.

Um den "schlafenden" Requester zu löschen, gibt es auch eine Funktion, die das für uns übernimmt. Sie heißt ClearDMRequest und benötigt lediglich einen Zeiger auf das Fenster, dessen Requester-Liste gelöscht werden soll.

ClearDMRequest	=	**-48 (Intuition-Library)**

***Window** **a0** < Zeiger auf eine Window-Struktur.

Erklärung Durch ClearDMRequest wird der gesetzte DMRequester wieder gelöscht.

Nun zum Demoprogramm:

```
*Programm 5.17: Anwendung von DM-Requestern

ExecBase      =       4
OldOpenLib    =       -408
CloseLib      =       -414
ReplyMsg      =       -378
WaitPort      =       -384
GetMsg        =       -372
OpenWindow    =       -204
CloseWindow   =       -72
SetDMRequest  =       -258
ClearDMRequest        =              -48

Start:  move.l   ExecBase,a6

        lea      IntName,a1
        jsr      OldOpenLib(a6)
        move.l   d0,IntBase
        beq      IntError

        move.l   IntBase,a6
        lea      WindowArgs,a0
        jsr      OpenWindow(a6)
        move.l   d0,WindowHD
        beq      WinError

        move.l   WindowHD,a0
        move.l   86(a0),UPort

        move.l   IntBase,a6
        lea      Requester,a1
        move.l   WindowHD,a0
        jsr      SetDMRequest(a6)
```

```
MessageLoop:
            move.l  ExecBase,a6

            move.l  UPort,a0
            jsr     GetMsg(a6)
            move.l  d0,Message
            bne     MessageBranch

            move.l  UPort,a0
            jsr     WaitPort(a6)
            bra     MessageLoop
Exit:       move.l  IntBase,a6
            move.l  WindowHD,a0
            jsr     ClearDMRequest(a6)

            move.l  IntBase,a6
            move.l  WindowHD,a0
            jsr     CloseWindow(a6)

WinError:
            move.l  ExecBase,a6
            move.l  IntBase,a1
            jsr     CloseLib(a6)
IntError:
            rts

MessageBranch:
            move.l  Message,a0
            move.l  20(a0),Class
            move.l  28(a0),a0
            move.w  38(a0),GadID

            cmp.l   #$800,Class
            bne     NoReq
            bset    #1,Flag

NoReq:
            move.l  Message,a1
            jsr     ReplyMsg(a6)

            btst    #1,Flag
            bne     ReqMessage

            cmp.l   #$200,Class
            beq     Exit

            bra     MessageLoop

ReqMessage:
            cmp.l   #$1000,Class
            bne     OhNo
            bclr    #1,Flag

OhNo:       bra     MessageLoop
```

```
* Datenbereich

GadID:          dc.w    0
Class:          dc.l    0
Flag:           dc.w    0
IntBase:        dc.l    0
WindowHD:       dc.l    0
Message:        dc.l    0
UPort:          dc.l    0

IntName:        dc.b    "intuition.library",0
                even
WinName:        dc.b    "Window",0
                even

WindowArgs:
                dc.w    0,0,640,256
                dc.b    1,3
                dc.l    $1A00,$0100F,0,0,WinName,0,0
                dc.w    100,50,200,100,1

Requester:      dc.l    0
                dc.w    0
                dc.w    0
rq_Width:       dc.w    38*8
rq_Height:      dc.w    54
rq_RelLeft      dc.w    -275
rq_RelTop       dc.w    -38
rq_ReqGadget:   dc.l    EndGadget
rq_ReqBorder:   dc.l    Border0
rq_ReqText:     dc.l    IText0
rq_Flags:       dc.w    1
                dc.b    0
                even
                dc.l    0
                dcb.b   32,0
                dc.l    0
                dc.l    0
                dcb.b   36,0

* Gadget

EndGadget:      dc.l    0
                dc.w    260,28,32,20,4,5,1
                dc.l    Image0,0,0,0,0
                dc.w    1
                dc.l    0

* IntuiText-Struktur

IText0:         dc.b    1,3,0
                even
                dc.w    21,11
                dc.l    0,ITextData0,IText1

ITextData0:
```

```
                dc.b    "Dies ist ein DM-Requester !",0
                even

IText1:         dc.b    2,3,0
                even
                dc.w    20,10
                dc.l    0,ITextData0,0

* Border-Data

Border0:        dc.w    0,0
                dc.b    2,0,0,3
                dc.l    BorderData0,Border1
BorderData0:
                dc.w    0,52,0,0,299,0

Border1:        dc.w    0,0
                dc.b    1,0,0,3
                dc.l    BorderData1,0

BorderData1:
                dc.w    299,1,299,52,2,52

* Image-Data

Image0:         dc.w    0,0,32,20,2
                dc.l    ImageData0
                dc.b    %11,0
                dc.l    0

                Section "",Data_C

ImageData0:
                dc.l    $00000000,$00000001,$1FFFFFF9,$10000001
                dc.l    $10000001,$10000001,$10000001,$101FFC01
                dc.l    $101FFC01,$101FFC01,$101FFC01,$101FFC01
                dc.l    $101FFC01,$103FFC01,$10000001,$10000001
                dc.l    $10000001,$10000001,$00000001,$FFFFFFFF

                dc.l    $FFFFFFFF,$80000000,$80000008,$80000008
                dc.l    $80000008,$80000008,$803FFC08,$803FF808
                dc.l    $803FF808,$803FF808,$803FF808,$803FF808
                dc.l    $80200008,$80000008,$80000008,$80000008
                dc.l    $80000008,$8FFFFFF8,$80000000,$80000000
```

Programm 5.17: Anwendung von DM-Requestern

5.7 Auswertung sonstiger Nachrichten

Nachdem wir fast am Ende dieses Kapitels angekommen sind, wollen wir uns nochmals der IntuiMessage-Struktur zuwenden und auf spezielle Arten von Nachrichten, die wir durch sie empfangen können, eingehen.

5.7.1 RAWKEY/VANILLAKEY

Wie wir schon wissen, müssen wir, wenn wir über Tastatureingabe in unser Fenster benachrichtigt werden wollen, eins der beiden IDCMP-Flags VANILLAKEY oder RAWKEY setzen. Bei VANILLAKEY können wir aus der IntuiMessage-Struktur den schon behandelten Tastencode auslesen. Das heißt, wir erhalten das Zeichen, welches der gedrückten Taste durch eine Tastaturtabelle zugeordnet worden ist.

Bei RAWKEY erhalten wir zwar auch einen Wert, es handelt sich hierbei jedoch um den unbehandelten, also rohen Tastaturcode (RAW-Keycode).

ESC $45	F1 $50	F2 $52	F3 $52	F4 $53	F5 $54	F6 $55	F7 $56	F8 $57	F9 $58	F10 $59

' $00	1 ! $01	2 @ $02	3 # $03	4 $ $04	5 % $05	6 ^ $06	7 & $07	8 * $08	9 ($09	0) $0A	- _ $0B	= + $0C	\ $0D	BACKS $41
TAB $42	Q q $10	W w $11	E e $12	R r $13	T t $14	Y y $15	U u $16	I i $17	O o $18	P p $19	[{ $1A	] } $1B	RETURN	
CTRL $63	CAPSL $62	A a $20	S s $21	D d $22	F f $23	G g $24	H h $25	J j $26	K k $27	L l $28	; : $29	" ' $2A	$2B	$44
SHIFT $60	| $30	Z z $31	X x $32	C c $33	V v $34	B b $35	N n $36	M m $37	, < $38	. > $39	/ ? $3A	SHIFT $61		
ALT $64	A $66	SPACE $40	A $67	ALT $65										

DEL $46	HELP $5F

	UP $4C	
LEFT $4F	DOWN $4D	RIGTH $4E

{ [$1A	}] $1B	/ $3A	* $88
7 $3D	8 $3E	9 $3F	- $4A
4 $2D	5 $2E	6 $2F	+ $0C
1 $1D	2 $1E	3 $1F	ENTER $43
0 $0F		DEL $3C	

Bild 5.12: Tastaturbelegung mit RAW-Codes

Jede Taste hat einen sogenannten RAW-Code und kann so einzeln abgefragt werden. Neben dem Wert der gedrückten Taste, der übrigens im Eintrag Code der IntuiMessage-Struktur steht, bekommen wir auch noch mitgeteilt, ob eine Qualifier-Taste gedrückt worden ist. Unter Qualifier-Tasten versteht man z.B. die Shift- oder Control-Taste. Wir können die Flags der Tasten im Eintrag Qualifier der IntuiMessage-Struktur überprüfen.

Folgende Qualifier-Flags stehen zur Verfügung:

```
IEQUALIFIER_LSHIFT          =     $0001
IEQUALIFIER_RSHIFT          =     $0002
IEQUALIFIER_CAPSLOCK        =     $0004
IEQUALIFIER_CONTROL         =     $0008
IEQUALIFIER_LALT            =     $0010
IEQUALIFIER_RALT            =     $0020
IEQUALIFIER_LCOMMAND        =     $0040
IEQUALIFIER_RCOMMAND        =     $0080
IEQUALIFIER_NUMERICPAD      =     $0100
IEQUALIFIER_REPEAT          =     $0200
IEQUALIFIER_INTERRUPT       =     $0400
IEQUALIFIER_MULTIBROADCAST  =     $0800
IEQUALIFIER_MIDBUTTON       =     $1000
IEQUALIFIER_RBUTTON         =     $2000
IEQUALIFIER_LEFTBUTTON      =     $4000
IEQUALIFIER_RELATIVEMOUSE   =     $8000
```

5.7.2 MOUSEBUTTONS/MOUSEMOVE

Sicher erinnern Sie sich auch noch an das IDCMP-Flag MOUSEBUTTONS. Es gab Intuition die Erlaubnis, eine Nachricht zu senden, sobald eine Maustaste gedrückt worden ist. Dabei ist jedoch noch nicht zu erkennen, welche Maustaste betätigt worden ist. Diese Information können wir auch aus der IntuiMessage-Struktur auslesen, dazu müssen wir nur den Code-Eintrag auf folgende Werte überprüfen:

```
SELECTDOWN  =  $0068      ; LMT gedrückt
SELECTUP    =  $00E8      ; LMT losgelassen
MENUDOWN    =  $0069      ; RMT gedrückt
MENUUP      =  $00E9      ; RMT losgelassen
```

Achtung!: Man erhält für die rechte Maustaste nur eine Meldung, wenn das Flag RMBTRAP gesetzt worden ist.

Die Mausbewegung ist noch einfacher. Man wertet einfach die Einträge MouseX und MouseY der IntuiMessage-Struktur aus, wenn man eine Meldung des Typs MOUSEMOVE erhalten hat.

5.8 Sonstige Funktionen

Bevor wir die Struktur der IntuitionBase besprechen, folgen nun alle Library-Funktionen, die noch nicht besprochen wurden.

AlohaWorkbench	=	**-402 (Intuition-Library)**

WBPort	**a0**	<	Zeiger auf den WB-MessagePort.
Erklärung			Nach Ausführen dieser Funktion meldet Intuition an den angegebenen MessagePort eine Meldung vom Typ WBMESSAGE, wenn die Workbench vom eigenen oder anderen Programmen geschlossen werden soll, damit man Gelegenheit erhält, Fenster-Aufräumarbeiten zu erledigen.

CurrentTime	=	**-84 (Intuition-Library)**

***Seconds**	**a0**	<	Speicher für Sekundenwert
***Micros**	**a1**	<	Speicher für Mikrosekundenwert
Erklärung			Durch die Funktion CurrentTime wird die aktuelle Systemzeit in die angegebenen Variablen kopiert.

DoubleClick	=	**-102 (Intuition-Library)**

SSeconds	**d0**	<	Sekundenwert der Startzeit
SMicros	**d1**	<	Mikrosekundenwert der Startzeit
CSeconds	**d2**	<	Sekundenwert der aktuellen Zeit
CMicros	**d3**	<	Mikrosekundenwert der aktuellen Zeit
isdouble	**d0**	>	-1, falls das Zeitintervall ein DoubleClick war
Erklärung			Prüft, ob das angegebene Zeitintervall Startzeit - aktuelle Zeit kurz genug war, um laut den Preferences-Einstellungen als DoubleClick zu gelten.

GetDefPrefs	=	**-126 (Intuition-Library)**

***Preferences**	**a0**	<	Zeiger auf Speicherbereich, in dem die Preferencesdaten abgelegt werden sollen (nähere Beschreibung finden sie im Anhang).

Size	**d0**	<	Größe des Puffers
Prefs	**d0**	>	Zeiger auf den Puffer, in dem die Daten abgelegt worden sind. Hier sollte der in a0 gespeicherte Zeiger übergeben werden.
Erklärung			Durch die Funktion GetDefPrefs (GetDefaultPreferences) werden die Standardvoreinstellungen ausgelesen und in den angegebenen Puffer abgelegt.

GetPrefs = -132 (Intuition-Library)

***Preferences**	**a0**	<	Zeiger auf Speicherbereich in dem die Preferencesdaten abgelegt werden sollen (nähere Beschreibung finden sie im Anhang).
Size	**d0**	<	Größe des Puffers
Prefs	**d0**	>	Zeiger auf den Puffer, in dem die Daten abgelegt worden sind. Hier sollte der in a0 übergebene Zeiger übergeben werden.
Erklärung			Durch die Funktion GetPrefs werden die aktuellen Voreinstellungen ausgelesen und in den angegebenen Puffer abgelegt.

ViewAddress = -294 (Intuition-Library)

View	**d0**	>	Nachdem die Funktion aufgerufen wurde, erhält man in d0 einen Zeiger auf die View-Struktur von Intuition wieder.
Erklärung			Durch die Funktion ViewAddress erhält man die Adresse der Intuition-View-Struktur (näheres siehe Graphics-Kapitel).

ViewPortAddress = -300 (Intuition-Library)

***Window**	**a0**	<	Zeiger auf ein Fenster
View	**d0**	<	Zeiger auf den ViewPort, der für das Fenster verantwortlich ist.
Erklärung			Mit der Funktion ViewPortAddress kann man die Adresse der ViewPort-Struktur des angegebenen Fensters ermitteln (näheres siehe Graphics-Kapitel).

SetPrefs	=	**-324 (Intuition-Library)**

***Prefbuffer**	**a0**	<	Zeiger auf Puffer, aus dem die Preferences-Daten geholt werden sollen
Size	**d0**	<	Anzahl zu kopierender Prefs-Bytes
Flags	**d1**	<	-1, falls Window-Flag NEWPREFS gesetzt werden soll

Erklärung

Setzt die System-Preferences neu, indem ihre Daten aus einem Puffer (Prefbuffer) in den System-Prefs-Bereich kopiert werden. Wahlweise kann danach an das Fenster die Message NEWPREFS geschickt werden.

LockIBase	=	**-414 (Intuition-Library)**

lock	**d0**	>	Locknummer für UnlockIBase

Erklärung

Verhindert Scheibzugriffe auf die Intuition-Base-Struktur durch die Int-Library (Struktur siehe weiter unten). Dadurch wird ein Umschalten von Screens, Windows usw. und auch die Bewegung der Maus verhindert.

UnLockIBase	=	**-420 (Intuition-Library)**

iblock	**a0**	>	Von LockIBase erhaltene Lock-Adresse

Erklärung

Hebt die Intuition-Base-Sperre wieder auf. Achtung: Eine falsche Lock-Adresse führt zum Absturz!

5.9 Die Basis-Struktur der Intuition-Library

Bisher haben wir die Basisadresse der Intuition-Library immer dazu benutzt, um mit negativen Offsets auf ihre Funktionen zuzugreifen. Doch die IntuitionBase-Struktur besteht nicht nur aus der Sprungtabelle, sondern hat auch noch einen Datenteil, der mit positiven Offsets angesprochen werden kann. Man findet hier ganz interessante Einträge. Deshalb sehen wir uns zum Abschluß auch diese Struktur an.

IntuitionBase-Struktur:

```
000      dc.l    *ln_Succ           ;┐           ┐
004      dc.l    *ln_Pred           ;│           │
008      dc.b     ln_Type           ;│ Node      │
009      dc.b     ln_Pri            ;│           │
010      dc.l    *ln_Name           ;┘           │
014      dc.w     lib_Flags         ;            │
016      dc.w     lib_NegSize       ;            │  Library-Struktur
018      dc.w     lib_PosSize       ;            │
020      dc.w     lib_Version       ;            │
022      dc.w     lib_Revison       ;            │
024      dc.l    *lib_idString      ;            │
028      dc.l     lib_Sum           ;            │
032      dc.w     lib_OpenCnt       ;            ┘
034      dc.l    *ib_ViewPort       ; Erster Viewport
038      dc.l    *ib_LOFCprList     ; Haupt-Copperliste
042      dc.l    *ib_SHFCprList     ; Interlace-Zweitcopperliste
046      dc.w     ib_DyOffset       ; y-Offset des Intui-View
048      dc.w     ib_DxOffset       ; x-Offset des Intui-View
050      dc.w     ib_Modes          ; Erlaubte Viewmodes
052      dc.l    *ib_ActiveWindow   ; aktives Fenster
056      dc.l    *ib_ActiveScreen   ; aktiver Screen
060      dc.l    *ib_FirstScreen    ; erster Screen
064      dc.l     ib_Flags
068      dc.w     ib_MouseY         ; Y-Mausposition
070      dc.w     ib_MouseX         ; X-Mausposition
072      dc.l     ib_Seconds        ; Systemzeit
076      dc.l     ib_Micros         ;

         ; Der Datenteil der Intuition-Library ist zwar an dieser
         ; Stelle noch nicht beendet, doch sind die Werte, die jetzt
         ; folgen, nicht festgelegt.
```

***ln_Succ - lib_OpenCnt**
Die ersten dreizehn Einträge sind für die Verwaltung der Library reserviert. Ihre Bedeutungen werden erst im Exec-Kapitel beschrieben.

***ib_ViewPort**
Zeiger auf die erste ViewPort-Struktur des Intuition-View.

***ib_LOFCprList**
Zeiger auf Copperliste für die Auflösungen interlaced und nicht-interlaced.

***ib_SHFCprList**
Zeiger auf Copperliste für interlaced-Darstellung.

ib_DyOffset, ib_DxOffset
Positionierung des Intuition-View. Es sind auch negative Werte erlaubt (siehe auch Graphics-Kapitel, Programm "BenchQuake").

ib_Modes
Erlaubte ViewModes.

***ib_ActiveWindow**
Zeiger auf Fenster-Struktur des aktiven Fensters.

***ib_ActiveScreen**
Zeiger auf Screen-Struktur des aktiven Screens.

***ib_FirstScreen**
Zeiger auf erste Screen-Struktur.

ib_Flags
Intuition-interne Flags.

ib_MouseY, ib_MouseX
Position des Mauszeigers.

ib_Seconds, ib_Micros
Systemzeit im Intuition-Format (Sekunden/Mikrosekunden).

Kapitel 6
Die Diskfont-Library

Grundwissen über Diskfonts

Einladen und Entfernen von Fonts

Verfügbare Zeichensätze

Textausgabe mit Diskfonts

Insider-Wissen

Die Diskfont-Library ist für die Verwaltung der Zeichensätze zuständig, die nicht im ROM, sondern auf der Diskette untergebracht sind. Da beim Amiga nur der System-Font ("Topaz") im ROM steht, wird man um die Benutzung der Diskfont-Library wohl kaum herumkommen.

Da es langweilig wäre, bei dem großen Angebot an Zeichensätze für den Amiga nur den Topaz-Font zu benutzen (die Designer der übrigen Fonts haben sich schließlich auch Arbeit gemacht), wollen wir in diesem Kapitel lernen, wie man Zeichensätze von der Diskette einlädt, in eigenen Programmen benutzt und wieder aus dem Speicher entfernt. Außerdem werden wir erfahren, wie man sich Informationen über die verfügbaren Fonts verschafft. Anschließend werden wir uns etwas Hintergrundwissen über den Aufbau einer Font-Datei aneignen.

Die Diskfont-Library ist nicht im Kickstart-ROM, sondern auf der Diskette untergebracht, weshalb beim ersten Öffnen die Startdiskette verlangt wird. Die Lib muß aber nur einmal eingeladen werden und bleibt dann bis zum nächsten Reset im RAM.

6.1 Grundwissen über Fonts

Alle Disk-Zeichensätze sind im FONTS-Verzeichnis untergebracht, in dem sie von allen Diskfont-Routinen auch gesucht werden. Hinweis: Gesucht wird im logischen Gerät FONTS:, welches dem FONTS-Verzeichnis der Startdiskette zugewiesen ist. Sollen Zeichensätze von einer anderen Diskette geladen werden, so muß die FONTS:-Zuweisung mit dem CLI-Befehl ASSIGN geändert werden.

Für jeden Font wird im FONTS-Verzeichnis ein Verzeichnis mit dem Namen des Fonts (z.B. "garnet") und eine Verwaltungsdatei, die das Anhängsel ".font" bekommt (z.B. "garnet.font"), angelegt.

Das Amiga-System sieht die Möglichkeit vor, ein Font in mehreren Größen zu verwenden. Als Kriterium für die Größeneinteilung wird immer die Höhe, also die Vertikalgröße, benutzt. Den Garnet-Font z.B. gibt es in zwei Größen: 16 und 9 (also 16 und 9 Punkte hoch). Die Font-Daten für die verschiedenen Größen werden jeweils in einer eigenen Datei abgelegt. Diese Dateien werden in den Font-Unterverzeichnisse abgespeichert und bekommen als Namen die Fontgröße. Im Unterverzeichnis "garnet" existieren also zwei Dateien mit den Namen "16" und "9".

In den ".font"-Dateien sind nun lediglich Informationen über die verfügbaren Größen eines Fonts sowie über diverse andere Eigenheiten abgelegt. Die Zeichendaten selber stehen in den Unterverzeichnis-Dateien. Das dient dazu, daß zum Einlesen

aller verfügbaren Fonts nicht sämtliche Unterverzeichnisse und die darin enthaltenen Dateien durchgelesen werden müssen, sondern nur die kleinen Info-Dateien im Fonts-Hauptverzeichnis.

6.2 Das Einladen von Fonts

Zum Laden eines Fonts von der Diskette dient die Routine OpenDiskFont:

OpenDiskFont	=	-30 (Diskfont-Library)

***textAttr**	**a0**	<	Zeiger auf TextAttr-Struktur, die den gesuchten Zeichensatz beschreibt
***font**	**d0**	>	Zeiger auf TextFont-Struktur des geladenen Font oder 0 bei Fehler
Erklärung			Lädt den in der TextAttr-Struktur definierten Zeichensatz von Diskette

Wenn der Font geladen werden konnte, haben wir in d0 den Zeiger auf eine TextFont-Struktur. Diesen Zeiger müssen wir bei allen weiteren Operationen mit dem Zeichensatz angeben. Zum Öffnen eines Fonts, der im ROM steht oder schon einmal von Diskette geladen wurde, gibt es noch eine andere Routine. Diese steht allerdings in der Graphics-Library und kann nicht auf die Diskette zugreifen. Ihr Name ist OpenFont. Ihre Parameter und Funktionsweise entsprechen bis auf die Einschränkung 'kein Disk-Zugriff' exakt der von OpenDiskFont. Vorteile bietet die Benutzung von OpenFont nicht, da auch OpenDiskFont den Zeichensatz nur einmal wirklich einlädt und allen weiteren Programmen, die ihn öffnen wollen, nur noch den TextFont-Zeiger übermittelt.

OpenDiskFont erwartet einen Zeiger auf eine neue Struktur, nämlich TextAttr. TextAttr steht für "Text Attributes", die Struktur beschreibt also die Eigenschaften des gewünschten Zeichensatzes. Sie sieht folgendermaßen aus:

Die TextAttr-Struktur:

```
00    dc.l   *ta_Name           ; Zeiger auf Font-Name
04    dc.w    ta_YSize          ; Vertikale Größe des Fonts
06    dc.b    ta_Style          ; Stil des Fonts (siehe unten)
07    dc.b    ta_Flags          ; Art des Fonts
08            ta_SIZEOF
```

***ta_Name**
Ein Zeiger auf den Namensstring des Zeichensatzes, der auch die Endung ".font" enthält und wie fast alle Namenstexte mit einem Nullbyte abgeschlossen sein muß.

ta_YSize
Die Höhe des gesuchten Fonts. Falls es den Font in dieser Höhe nicht gibt, wird die Höhe benutzt, die der gewünschten am nächsten kommt.

ta_Style
Bit-Maske für den Schreibstil des Fonts. Folgende Werte, die bei Bedarf natürlich auch kombiniert (aufaddiert) werden können, sind möglich:

Schreibstil	Wert	Bedeutung
FSF_NORMAL	0	Kein besonderer Stil
FSF_UNDERLINED	1	Unterstrichen
FSF_BOLD	2	Fettdruck
FSF_ITALIC	4	Kursiv-(Schräg-)Druck
FSF_EXTENDED	8	Doppelte Breite (bei normalen Fonts nicht möglich)

ta_Flags
Bit-Maske für diverse andere Zeichensatz-Eigenarten. Hier die equ-Tabelle:

Font-Flag	Wert	Bedeutung
FPF_ROMFONT	1	Font ist im ROM
FPF_DISKFONT	2	Font wurde von Disk geladen
FPF_REVPATH	4	Ausrichtung von rechts nach links
FPF_TALLDOT	8	Hires-Noninterlaced-Font
FPF_WIDEDOT	16	Lores-Interlaces-Font
FPF_PROPORTIONAL	32	Proportional-Font
FPF_DESIGNED	64	Font ist gezeichnet, nicht berechnet
FPF_REMOVED	128	Font wurde per Lib-Routine entfernt

FPF_ROMFONT Der Zeichensatz befindet sich im ROM (gilt z.Z. nur für den Topaz-Font).

FPF_DISKFONT Der Zeichensatz ist auf Diskette zu finden.

FPF_REVPATH Die Zeichen sollen nicht von links nach rechts, sondern von rechts nach links ausgegeben werden.

FPF_TALLDOT Der Zeichensatz ist für eine Bildschirmauflösung von 640x256 Punkten (Hires Non-Interlaced) gedacht.

FPF_WIDEDOT Der Zeichensatz ist für eine Bildschirmauflösung von 320x512 Punkten (Lores Interlaced) gedacht.

FPF_PROPORTIONAL Die Zeichen werden auf dem Bildschirm proportional ausgegeben, d.h. sie nehmen nur so viel Platz ein, wie sie wirklich brauchen. Bei nicht-proportional-Zeichensätzen nehmen alle Zeichen gleich viel Platz ein.

FPF_DESIGNED	Die Zeichensatzgröße wurde gesondert entworfen und nicht vom System berechnet. Eine solche Berechnung ist erst ab der Version 2.0 des Betriebssystems möglich.
FPF_REMOVED	Der Zeichensatz wurde mit einer Graphics-Routine aus der Systemliste entfernt.

Der Eintrag ta_Flags ist nur vollständig von Interesse, wenn Sie sich Informationen über alle verfügbaren Zeichensätze geben lassen. Beim Öffnen eines bestimmten Fonts brauchen nur FPF_ROMFONT bzw. FPF_DISKFONT entsprechend gesetzt zu werden.

Zum Eintrag ta_YSize ist noch folgendes zu sagen. Unter Kickstart 2.0 ist es möglich, einen Zeichensatz auch in einer Größe darzustellen, die nicht eigens entworfen wurde. Gibt man in ta_YSize eine Größe an, die im Font-Verzeichnis nicht existiert, wird der Font, der dieser Größe am nächsten kommt geladen und dann auf die gewünschte Größe umgerechnet. Die Punkte, aus denen die Zeichen aufgebaut sind, werden dabei einfach je nach Bedarf verdoppelt, verdreifacht usw. Die Zeichen sind danach zwar größer, sehen aber auch viel grober aus. Bei älteren Versionen des Betriebssystems ist eine solche automatische Vergrößerung noch nicht möglich, hier wird bei nicht vorhandener Größe nur der Font geladen, der der gewünschten Größe am nächsten kommt.

Die TextFont-Struktur (nicht zu verwechseln mit TextAttr), auf die der Rückgabewert von OpenDiskFont zeigt, ist für den Programmierer eigentlich weniger wichtig. Sie wird nur vom System zur Verwaltung des Fonts benutzt. Sie soll hier aber trotzdem vorgestellt werden:

Die TextFont-Struktur:

```
00   ds.b   tf_Message,20      ; Eine Exec-Message (siehe unten)
20   dc.w   tf_YSize           ; Y-Größe des Fonts
22   dc.b   tf_Style           ; Schreibstil
23   dc.b   tf_Flags           ; Font-Eigenschaften
24   dc.w   tf_XSize           ; Standard-Fontbreite
26   dc.w   tf_Baseline        ; Grundlinie
28   dc.w   tf_BoldSmear
30   dc.w   tf_Accessors       ; Anzahl der Zugriffe
32   dc.b   tf_LoChar          ; Erstes ASCII-Zeichen im Font
33   dc.b   tf_HiChar          ; Letztes ASCII-Zeichen im Font
34   dc.l  *tf_CharData        ; Zeiger auf die Zeichen-Daten
38   dc.w   tf_Modulo
40   dc.l  *tf_CharLoc
44   dc.l  *tf_CharSpace       ; Zeiger auf Proportional-Daten
48   dc.l  *tf_CharKern        ; Zeiger auf Kerning-Daten
52          tf_SIZEOF
```

tf_Message

Vor den eigentlichen Font-Daten befindet sich eine Struktur, die aus der Exec-Library stammt. Sie nennt sich "Message-

Struktur" und wird im Kapitel über die Exec-Lib genauer besprochen.

tf_YSize, tf_Style, tf_Flags
Diese drei Einträge entsprechen den gleichnamigen in der TextAttr-Struktur.

tf_XSize
Bei nicht-proportionalen Fonts steht hier die Pixelbreite der Zeichen. Bei Proportionalfonts hat der Eintrag keine Bedeutung.

tf_BaseLine
Hier wird der Abstand der Grundlinie vom oberen Zeichenrand eingetragen. Auf diese Linie beziehen sich die y-Positionen bei Textausgaben. Wenn ein Text z.B. ab der Zeile 200 ausgegeben werden soll, seine Grundlinie aber 10 Pixel von der Oberkante entfernt liegt, wird er in Wirklichkeit ab Zeile 190 ausgegeben.

tf_Accessors
Die Anzahl der OpenDiskFont-Aufrufe für den Zeichensatz. Beim Schließen des Zeichensatzes wird der Zähler um eins verkleinert. Wenn er Null erreicht, wird der Zeichensatz aus dem Speicher gelöscht.

tf_LoChar, tf_HiChar
Das erste und letzte ASCII-Zeichen, für das der Zeichensatz entworfen wurde.

***tf_CharData**
Ein Zeiger auf die eigentlichen Pixel-Daten des Fonts.

***tf_CharSpace, *tf_CharKern**
Zwei Zeiger auf Datenfelder, die bei Proportionalfonts benutzt werden. In den Feldern stehen die Pixelbreiten der einzelnen Zeichen.

Wenn ein Zeichensatz nicht mehr benötigt wird, sollte er geschlossen werden. Das übernimmt die Routine CloseFont, die sich allerdings in der Graphics-Library befindet. Wir greifen hier etwas vor, aber die CloseFont-Routine paßt thematisch besser in dieses Kapitel.

CloseFont	=	**-78 (Graphics-Library)**

***textFont**	a1	<	Zeiger auf die TextFont-Struktur des zu schließenden Zeichensatzes
Erklärung			Schließt den angegebenen Zeichensatz. Wenn alle Öffnungs-Aufrufe durch den Close-Aufruf rückgängig gemacht wurden, wird der Zeichensatz aus dem Speicher entfernt.

Noch ein kurzes Beispielprogramm, das den Zeichensatz "Garnet" mit einer Y-Größe von 16 Punkten öffnet und wieder schließt. Die Basis der Graphics- und Diskfont-Library werden in Variablen vorausgesetzt.

```
        ...
        move.l  dfbase,a6       ; Diskfont-Basis nach a6
        lea     textattr,a0     ; Zeiger auf TextAttr-Struktur
        jsr     -30(a6)         ; Font öffnen
        move.l  d0,d6           ; TextFont-Zeiger sichern
        ...
        ...
        move.l  gfxbase,a6      ; Graphics-Basis nach a6
        move.l  d6,a1           ; TextFont-Zeiger nach a1
        jsr     -78(a6)         ; Font schließen
        ...

textattr:     dc.l    fontname        ; Zeiger auf Font-Namenstext
              dc.w    16              ; Höhe: 16 Pixel
              dc.b    0               ; Text-Stil: Normal
              dc.b    2               ; Font-Art: Diskfont
fontname:     dc.b    "garnet.font",0
              even
```

Bild 6.1: Öffnen und Schließen eines Fonts

Nun wissen wir, wie man Zeichensätze öffnet und schließt. Bevor wir aber zur ihrer Benutzung in Programmen kommen, wollen wir uns erst ansehen, wie man sich Informationen über alle verfügbaren Zeichensätze geben lassen kann.

6.3 Informationen über verfügbare Zeichensätze

In der Diskfont-Library existiert eine Routine, die einen Pufferspeicher, den wir reservieren und dessen Startadresse wir der Routine übergeben müssen, mit diversen Strukturen füllt. Aus diesen Strukturen kann man Informationen über alle auf der Diskette (und im ROM) verfügbaren Fonts, ihre Größen, Schriftarten und sonstigen Eigenheiten ablesen. Die Routine heißt AvailFonts:

AvailFonts	**=**	**-36 (Diskfont-Library)**

***buffer**	**a0**	<	Startadresse, ab der die Info-Strukturen angelegt werden sollen
bufBytes	**d0**	<	Größe des Info-Puffers in Bytes
flags	**d1**	<	Geben an, welche Typen von Fonts gesucht werden sollen

error	d0	>	Anzahl der Info-Bytes, die nicht mehr in den Puffer paßten oder 0, wenn alles in Ordnung war
Erklärung			Füllt einen Speicherbereich mit Info-Strukturen über die verfügbaren Fonts. Falls der Bereich zu klein ist, werden nicht alle Fonts behandelt und die Anzahl der fehlenden Bytes gemeldet.

Zuerst die möglichen Werte für die Flags:

Font-Typ	Wert	Bedeutung
AFF_MEMORY	1	Suche Fonts im ROM
AFF_DISK	2	Suche Fonts auf der Diskette

AFF_MEMORY und AFF_DISK können auch kombiniert (aufaddiert) werden, dann werden Fonts im ROM und auf Disk gesucht. Nun zu den Strukturen, die im Info-Puffer angelegt werden. Zu Beginn kommt eine AvailFontsHeader-Struktur. Sie besteht eigentlich nur aus einem Wort-Eintrag, der angibt, wieviele AvailFonts-Strukturen danach folgen werden:

Die AvailFontsHeader-Struktur:

```
00      dc.w    afh_NumEntries          ; Anzahl der folgenden AF-Strukturen
02      ds.b    afh_AF,10               ; Die erste AvailFonts-Struktur
12      ds.b    afh_AF,10               ; Die zweite AvailFonts-Struktur
..      ....    .........               ; usw.
```

Die Größe dieser Struktur ist also nicht festgelegt. Sie beträgt 2 (für das NumEntries) plus Anzahl Fonts * 10.

Die AvailFonts-Struktur:

```
00      dc.w     af_Type                ; AFF_MEMORY oder AFF_DISK
02      dc.l    *af_Name,0              ;┐
06      dc.w     af_YSize               ;│ TextAttr-
08      dc.b     af_Style               ;│   Struktur
09      dc.b     af_Flags               ;┘
10               af_SIZEOF
```

Die AvailFonts-Struktur besteht aus einem Wort, das angibt, ob der Font im ROM steht (1) oder auf Disk (2), und einer TextAttr-Struktur, welche die eigentlichen Informationen enthält.

Das folgende Beispielprogramm wird die Benutzung von AvailFonts noch etwas klarer machen. Es gibt Namen, Größen und Modi (proportional oder nicht) aller Fonts auf der Diskette aus.

```
* Programm 6.1: Anzeige aller verfügbaren Diskfonts

ExecBase     =       4
OpenLib      =       -552
CloseLib     =       -414
AvailFonts   =       -36
Output       =       -60
Write        =       -48

        move.l  ExecBase,a6      ; DOS-Lib öffnen
        lea     dosname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,dosbase       ; Basis sichern

        lea     dfname,a1        ; Diskfont-Lib öffnen
        clr.l   d0
        jsr     -552(a6)
        move.l  d0,dfbase        ; Basis sichern

        move.l  dfbase,a6        ; Benutze Diskfont-Lib

* Fontinfo-Puffer füllen

        lea     fontbuff,a0      ; Zeiger auf Fontinfo-Puffer
        move.l  #2000,d0         ; 2000 Bytes sollten reichen
        move.l  #2,d1            ; Flags: Nur Diskfonts
        jsr     AvailFonts(a6)   ; Puffer füllen lassen

        move.l  dosbase,a6       ; Benutze jetzt DOS-Lib
        jsr     Output(a6)       ; Output-Handle holen
        move.l  d0,d4

        lea     fontbuff,a5      ; Start Fontbuffer in Arbeitsregister
        move.w  (a5)+,d6         ; Anzahl Fonts nach d6
        subq    #1,d6            ; DBRA läuft bis -1

        lea     text0,a0         ; Titeltext
        bsr     print            ; ausgeben

* Hauptschleife: Einen Fonteintrag ausgeben

main1:  addq    #2,a5            ; Type-Wort in AF-Struktur überspr.
        move.l  (a5)+,a0         ; Zeiger auf Font-Name nach a0
        bsr     print            ; Name ausgeben

        lea     text1,a0         ; Zwischentext
        bsr     print            ; ausgeben

        clr.l   d0               ; d0 long-löschen
        move.w  (a5)+,d0         ; Fonthöhen-Wort nach d0
        lea     dezbuff,a0       ; Start des Dez-Puffers
```

```
        bsr     dezascii        ; Zahl umrechnen
        lea     dezbuff,a0      ; Dez-Puffer-Inhalt
        bsr     print           ; ausgeben

        move.w  (a5)+,d0        ; Style- u. Flag-Byte nach d0
        btst    #5,d0           ; Bit 5 im Flag-Byte gesetzt?
        beq     main2           ; Wenn nein, kein Prop-Font

        lea     text2,a0        ; Sonst Text "Prop" ausgeben
        bsr     print

main2:  lea     text3,a0        ; Return-Zeichen
        bsr     print           ; ausgeben

        dbra    d6,main1        ; Zur Hauptschleife

        move.l  ExecBase,a6     ; Jetzt Exec-Lib benutzen

        move.l  dosbase,a1      ; DOS-Lib
        jsr     CloseLib(a6)    ; schließen
        move.l  dfbase,a1       ; Diskfont-Lib
        jsr     CloseLib(a6)    ; schließen
        rts                     ; und Ende


print:  movem.l d1-d3,-(sp)     ; SUB Textausgabe für DOS-Write
        move.l  a0,d2           ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l   d3              ;  d4 < Handle der Ausgabedatei
pr1:    addq    #1,d3
        tst.b   (a0)+
        bne     pr1
        subq    #1,d3

        move.l  d4,d1
        jsr     Write(a6)
        movem.l (sp)+,d1-d3
        rts

dezascii:                       ; SUB Umrechnung Dez-Zahl -> ASCII-Text
        movem.l d1/d2/a1,-(sp)  ;  d0 <- Dezimalzahl
        clr.b   d2              ; *a0 <- Pufferzeiger
        lea     values,a1

da1:    clr.b   d1

da2:    addq    #1,d1
        sub.l   (a1),d0
        bcc     da2
        add.l   (a1),d0
        subq    #1,d1

        tst.b   d1
        beq     da3
```

```
        add.b    #"0",d1
        move.b   d1,(a0)+
        moveq    #1,d2
        bra      da4

da3:    tst.b    d2
        beq      da4
        move.b   #"0",(a0)+

da4:    cmp.l    #1,(a1)
        beq      da5

        add.l    #4,a1
        bra      da1

da5:    move.b   #0,(a0)
        movem.l  (sp)+,d1/d2/a1
        rts

* Datenbereich

dosname:        dc.b    "dos.library",0
                even
dfname:         dc.b    "diskfont.library",0
                even
dosbase:        dc.l    0
dfbase:         dc.l    0
fontbuff:       ds.b    2000
text0:          dc.b    10,"Liste der verfügbaren Disk-Fonts:",10
                dc.b    "---------------------------------",10,10,0
text1:          dc.b    "  ",0
text2:          dc.b    "  (Prop)",0
text3:          dc.b    10,0
dezbuff:        ds.b    8
values:         dc.l    1000000000,100000000,10000000,1000000
                dc.l    100000,10000,1000,100,10,1
```

Programm 6.1: Anzeige aller verfügbaren Diskfonts

Die Benutzung der AvailFonts-Routine dürfte klar sein. Das Library-Öffnen und Output-Handle-Holen sowieso. Interessant wird es an der Stelle

```
        lea      fontbuff,a5     ; Start Fontbuffer in Arbeitsregister
        move.w   (a5)+,d6        ; Anzahl Fonts nach d6
        subq     #1,d6           ; DBRA läuft bis -1
```

Wir speichern den Start des Fontpuffers ins Adreßregister a5, das wir in Zukunft immer zum Zugriff auf den Puffer benutzen werden. Das erste Wort ist die Anzahl der Fonts, also der AvailFonts-Strukturen, die nach dem Wort folgen werden. Die Hauptschleife wird mit DBRA aufgebaut, weshalb wir 1 von der Fontanzahl subtrahieren müssen. In der Zeile

```
main1: addq     #2,a5           ; Type-Wort in AF-Struktur überspr.
```

erhöhen wir den Pufferzeiger um zwei, wodurch wir das Type-Wort, das in jeder AF-Struktur vor dem TextAttr-Teil steht, überspringen. Da wir sowieso nur Diskfonts einladen, ist eine Auswertung des Font-Typs überflüssig. Das nächste Langwort ist ein Zeiger auf den Fontnamen, der, für unsere Print-Routine sehr günstig, mit einem Nullbyte endet. Wir können Print also direkt aufrufen:

```
        move.l  (a5)+,a0        ; Zeiger auf Font-Name nach a0
        bsr     print           ; Name ausgeben
```

Das nächste Wort beinhaltet die Fontgröße, die wir nach Umrechnung durch DezAscii von Print ausgeben lassen können. In der Zeile

```
        move.w  (a5)+,d0        ; Style- u. Flag-Byte nach d0
```

holen wir die Bytes für Style und Flags auf einmal nach d0. Eigentlich brauchen wir nur die Flags, aber durch das MOVE.W können wir uns ein Erhöhen des Adreßzeigers um eins sparen. Der MOVE-Befehl schreibt das Flags-Byte ins unterste Byte des Registers. Wir können also das Bit Nr. 5 von d0, welches das PROPORTIONAL-Bit des Flagbytes enthält, testen:

```
        btst    #5,d0           ; Bit 5 im Flag-Byte gesetzt?
        beq     main2           ; Wenn nein, kein Prop-Font
```

War das Bit nicht gesetzt, wird die Ausgabe des Textes "Prop" übersprungen. Der Rest des Programms dürfte eigentlich klar sein.

6.4 Textausgabe mit eingeladenen Fonts

Es gibt zwei Möglichkeiten, zur Textausgabe der eingeladenen Fonts. Die erste beruht auf einer Routine aus der Graphics-Library, genannt SetFont, die einen wählbaren Zeichensatz einstellt. Da diese Routine aber nur im Zusammenhang mit weiteren Graphics-Routinen und -Strukturen einsetzbar ist, werden wir im Graphics-Kapitel darauf zurückkommen.

Die zweite Möglichkeit beruht auf Intuition-Strukturen, die das Einsetzen eines Font-Zeigers erlauben. Das sind die New-Screen-Struktur und die IntuiText-Struktur. Man kann also schon beim Öffnen eines neuen Screens angeben, daß ein bestimmter Zeichensatz verwendet werden soll. Dieser wird dann für alle Textausgaben im Screen (auch die Titelzeile usw.) und in allen entsprechenden Windows benutzt.

Die IntuiText-Struktur wird bekanntlich überall da eingesetzt, wo es um direkte Textausgabe (mit PrintIText) oder Angabe eines Gadget- oder Menütextes usw. geht. Durch Ver-

wendung eines Font-Zeigers kann man hier individuelle Fonts einstellen, die sich auch vom Standard-Font (beim Öffnen des Screens angegeben) unterscheiden können.

Wichtig ist dabei, daß erstens der entsprechende Font vor der Benutzung in Intui-Strukturen geöffnet worden sein muß, und daß zweitens, der Font-Zeiger ein Zeiger auf die TextAttr-Struktur sein muß, also auf die selbe Struktur, die wir zum Öffnen mit OpenDiskFont benutzten, nicht den Rückgabe-Zeiger (auf TextFont-Struktur) dieser Routine. Die Graphics-Routine, die wir später kennenlernen werden, erwartet allerdings einen TextFont-Zeiger.

Alle erforderlichen Routinen und Strukturen zum Screenöffnen und zur Fontbenutzung sind ja schon bekannt, daher können wir sofort ein Beispielprogramm bringen. Es öffnet einen Screen unter der Benutzung des Fonts 'Garnet 16', auf dem Screen ein Fenster, und in dieses wird mit dem Font 'Emerald 17' ein Text ausgegeben. Diese beiden Fonts finden Sie "serienmäßig" auf der Workbench-Diskette.

```
* Programm 6.2: Benutzung von Diskfonts in Screens und Windows

ExecBase      =       4
OpenLib       =       -552
CloseLib      =       -414
OpenDiskFont  =       -30
OpenScreen    =       -198
OpenWindow    =       -204
PrintIText    =       -216
WaitPort      =       -384
GetMsg        =       -372
ReplyMsg      =       -378
CloseWindow   =       -72
CloseScreen   =       -66
CloseFont     =       -78

        move.l   4,a6             ; Int-Lib öffnen
        lea      intname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,intbase

        lea      gfxname,a1       ; Graphics-Lib öffnen
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,gfxbase

        lea      dfname,a1        ; Diskfont-Lib öffnen
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,dfbase
        tst.l    d0               ; Fehler?
```

```
        beq      ende1            ; Wenn ja

        move.l   dfbase,a6        ; Benutze Diskfont-Lib

        lea      tattr1,a0        ; Font 'Garnet 16' öffnen
        jsr      OpenDiskFont(a6)
        tst.l    d0               ; Fehler?
        beq      ende2            ; Wenn ja
        move.l   d0,tfont1        ; TextFont-Zeiger sichern

        lea      tattr2,a0        ; Font 'Emerald 17' öffnen
        jsr      OpenDiskFont(a6)
        tst.l    d0               ; Fehler?
        beq      ende3            ; Wenn ja
        move.l   d0,tfont2        ; Zeiger sichern

        move.l   intbase,a6

        lea      nscreen,a0       ; Screen öffnen
        jsr      OpenScreen(a6)
        move.l   d0,wscreen

        lea      nwindow,a0       ; Window öffnen
        jsr      OpenWindow(a6)
        move.l   d0,window

        move.l   d0,a0            ; Zeiger nach a0
        move.l   $32(a0),a4       ; Rastport nach a4
        move.l   $56(a0),a5       ; Userport nach a5

        move.l   a4,a0            ; Text ausgeben mit PrintIText
        lea      itext,a1
        move.w   #0,d0            ; Position steht in der Struktur
        move.w   #0,d1
        jsr      PrintIText(a6)

        move.l   ExecBase,a6

        move.l   a5,a0            ; Warte auf Nachricht
        jsr      WaitPort(a6)
        move.l   a5,a0            ; Nachricht
        jsr      GetMsg(a6)       ; abholen
        move.l   d0,a1            ; Nachricht
        jsr      ReplyMsg(a6)     ; quittieren

        move.l   intbase,a6

        move.l   window,a0        ; Window schließen
        jsr      CloseWindow(a6)

        move.l   wscreen,a0       ; Screen schließen
        jsr      CloseScreen(a6)

        move.l   gfxbase,a6
```

```
        move.l  tfont2,a1      ; Fonts schließen
        jsr     CloseFont(a6)
ende3:  move.l  tfont1,a1
        jsr     CloseFont(a6)

ende2:  move.l  ExecBase,a6    ; Alle Libs schließen
        move.l  dfbase,a1
        jsr     CloseLib(a6)
ende1:  move.l  gfxbase,a1
        jsr     CloseLib(a6)
        move.l  intbase,a1
        jsr     CloseLib(a6)

        rts

* Datenbereich

intname:        dc.b    "intuition.library",0
                even
dfname:         dc.b    "diskfont.library",0
                even
gfxname:        dc.b    "graphics.library",0
                even
intbase:        dc.l    0
dfbase:         dc.l    0
gfxbase:        dc.l    0

tattr1:         dc.l    fname1
                dc.w    16
                dc.b    0,2
fname1:         dc.b    "garnet.font",0
                even

tattr2:         dc.l    fname2
                dc.w    17
                dc.b    0,2
fname2:         dc.b    "emerald.font",0
                even

nscreen:        dc.w    0,0,640,256,1 ; Screen-Dimensionen und -tiefe
                dc.b    0,1           ; Zeichenstifte
                dc.w    $8000,15      ; Modus HIRES, Typ CUSTOMSCREEN
                dc.l    tattr1        ; Zeiger auf TextAttr für Font
                dc.l    stitle        ; Zeiger auf Titel
                dc.l    0,0           ; Keine Gadgets und Custombitmap

stitle:         dc.b    "Garnet-Screen (Font = Garnet 16)",0
                even

nwindow:        dc.w    0,30,640,180  ; Window-Dimensionen
                dc.b    0,1           ; Zeichenstifte
                dc.l    $200,15       ; IDCMP: CLOSEWINDOW, Alle Gadgets
                dc.l    0,0           ; Keine Gadgets und Checkmark
                dc.l    wtitle        ; Zeiger auf Titel
wscreen:        dc.l    0             ; Zeiger auf Screen
                dc.l    0             ; Keine Custombitmap
```

```
            dc.w    200,100,640,200     ; Min- und Max-Größe
            dc.w    15                  ; Typ CUSTOMSCREEN

wtitle:     dc.b    "<- Close-Gadget klicken zum Beenden",0
            even

window:     dc.l    0
tfont1:     dc.l    0
tfont2:     dc.l    0

itext:      dc.b    1,0                 ; Farben
            dc.b    0,0                 ; Draw-Mode u. Adjust-Byte
            dc.w    10,30               ; Text-Position
            dc.l    tattr2              ; Zeiger auf TextAttr für Font
            dc.l    itextr              ; Zeiger auf Text
            dc.l    0                   ; Kein weiterer Text

itextr:     dc.b    "Dies ist ein Text in Emerald 17",0
            even
```

Programm 6.2: Benutzung von Diskfonts in Screens und Windows

Das Interessanteste ist diesmal in den Strukturen zu finden. In der Screen-Struktur heißt es jetzt

```
            ...
            dc.l    tattr1              ; Zeiger auf TextAttr für Font
            ...
```

was die Benutzung von Garnet 16 als Standard-Font bewirkt. Der Screen-Titel und alle Windows, die wir öffnen wollen, werden in diesem Font dargestellt. Beachten Sie, daß nicht der von OpenDiskFont gemeldete TextFont-Zeiger hier eingetragen werden muß, sondern der Zeiger auf die TextAttr-Struktur, die wir auch zum Öffnen des Fonts verwendet haben. Auf die selbe Weise verfahren wir in der IText-Struktur:

```
            ...
            dc.l    tattr2              ; Zeiger auf TextAttr für Font
            ...
```

Beachten Sie noch die Zeilen

```
        move.l  a5,a0           ; Warte auf Nachricht
        jsr     WaitPort(a6)
        move.l  a5,a0           ; Nachricht
        jsr     GetMsg(a6)      ; abholen
        move.l  d0,a1           ; Nachricht
        jsr     ReplyMsg(a6)    ; quittieren
```

Wir warten hier einfach auf eine Nachricht, holen sie ab und quittieren sie sofort. Auszuwerten brauchen wir sie nicht, da es sowieso nur die eine sein kann, die wir für das Fenster zugelassen haben (CLOSEWINDOW).

Damit dürfte die Anwendung von Diskfonts in eigenen Screens und ITexten klar sein.

6.5 Insider-Wissen über Diskfonts

Nun wollen wir uns noch etwas Hintergrundwissen aneignen. Für die Benutzung von Fonts sind die beiden folgenden Abschnitte nicht unbedingt nötig, aber für den einen oder anderen sind sie vielleicht interessant.

6.5.1 Font-Informationen auf eine etwas andere Weise

Neben der besprochenen Routine AvailFonts, die eine gleichnamige Struktur aufbaut, gibt es noch eine weitere Routine: NewFontContents. Sie sieht folgendermaßen aus:

NewFontContents	**=**	**-42 (Diskfont-Library)**

***fontsLock**	**d0**	<	Lock auf das gewünschte Font-Directory
***fontName**	**a1**	<	Zeiger auf den Namen der Font-Verwaltungsdatei
***fch**	**d0**	>	Adresse der FontContentsHeader-Struktur oder 0 bei Fehler während der Erstellung
Erklärung			Liest die Font-Daten aus der angegebenen Verwaltungsdatei und stellt sie in einer FontContentsHeader-Struktur zusammen

Mit dieser Routine kann man sich Informationen über einen bestimmten Font besorgen. In d0 muß ein Lock auf das gewünschte Font-Verzeichnis stehen (z.B. "df0:fonts/garnet") und in a1 ein Zeiger auf den Namen der Font-Verwaltungsdatei (z.B. "df0:fonts/garnet.font"). Im Gegensatz zu AvailFonts braucht man keinen Zeiger auf den Speicherbereich zu übergeben, in dem die Info-Struktur angelegt werden soll. NewFontContents wählt den Speicherplatz selber aus. Die FontContentsHeader-Struktur (kurz 'fch'), in der die Informationen zusammengestellt werden, sieht so aus:

Die FontContentsHeader-Struktur:

```
000    dc.w   fch_FileID          ; $0f00
002    dc.w   fch_NumEntries      ; Anzahl der folgenden fc-Strukturen
004    ds.b   fch_FH,260          ; Die erste FC-Struktur
264    ds.b   fch_FH,260          ; Die zweite FC-Struktur
...    ....   ..........          ; usw.
```

fch_FileID
Dies ist das erste Wort, das in der ".font"-Datei steht. Es kennzeichnet die Datei als Font-Verwaltungsdatei. Da NewFontContents im Prinzip diese Datei einfach nur einlädt, steht in der fch-Struktur auch als erstes dieses Kennungs-Wort.

Der Aufbau dieser Struktur läuft analog zur AvailFontsHeader-Struktur. Das zweite Wort gibt an, wieviele FontContents-Strukturen folgen werden, die so aussehen:

Die FontContents-Struktur:

```
000   ds.b   fc_FileName,256      ; Name der Fontdatei (0-terminiert)
256   dc.w   fc_YSize             ; Y-Größe
258   dc.b   fc_Style             ; Schreibstil
259   dc.b   fc_Flags             ; Font-Eigenschaften
260          fc_SIZEOF
```

Die Felder entsprechen bis auf fc_FileName denen der TextAttr-Struktur. Der FileName-Eintrag steht hier in der Struktur selber und wird nicht über einen Zeiger angesprochen. Der Eintrag für den Namen ist immer 256 Bytes groß, auch wenn der Name für gewöhnlich viel kürzer sein dürfte (dann sind die restlichen Zeichen mit Nullen gefüllt). Auch hat der Name hier einen anderen Aufbau: Im Falle des Fonts Garnet 16 würde hier nicht "garnet.font" stehen, sondern "garnet/16", also der Pfad- und Dateiname der Font-Datendatei (vom Verzeichnis FONTS: aus gesehen).

Da die Routine NewFontContents selbstständig einen Speicherbereich für die Struktur auswählt, muß es auch eine Routine geben, die diesen wieder freigibt. Sie heißt DisposeFontContents und sieht folgendermaßen aus:

DisposeFontContents = -48 (Diskfont-Library)

***fch**	a1	<	Zeiger auf die zu entfernende fch-Struktur
Erklärung			Gibt den von der angegebenen fch-Struktur belegten Speicher frei.

Sinnvoll ist die Benutzung von NewFontContents, wenn Sie nur über einen bestimmten Font Informationen benötigen, ansonsten benutzen Sie besser AvailFonts.

6.5.2 Der Aufbau einer Font-Datendatei

Ob Sie es glauben oder nicht, eine Font-Datendatei ist im Prinzip ein ausführbares Programm! Sie können es ja mal ver-

suchen und "fonts:garnet/16" als Programm starten. Eventuell müssen Sie vorher das Execute-Schutzbit mit dem CLI-Befehl SetProtection setzen. Passieren wird dabei allerdings nicht viel. Woran das liegt, werden Sie gleich erfahren.

Der Grund, warum die Font-Dateien den gleichen Aufbau haben wie ausführbare Programme, liegt in den Zeigern, die in der Fontdatei verwendet werden. Die Datei muß ja an jeden beliebigen Platz im Speicher geladen werden können, ein absoluter Zeiger würde sie aber an eine bestimmte Adresse binden. Um das zu umgehen, wird das Einladen vom DOS-Programmlader übernommen, der die absoluten Adressen bekanntlich anhand einer Tabelle anpaßt.

Damit es aber keinen Absturz gibt, wenn man eine Fontdatei "einfach mal aus Spaß" als Programm startet, stehen zu Beginn der Datei die Befehle "moveq #0,d0" und "rts" in der Datei. Vor diesen Befehlen findet sich noch ein Null-Zeiger. An seiner Stelle steht bei einem richtigen Programm der Zeiger auf das zweite Programmsegment (falls vorhanden), da es bei einer Fontdatei aber kein solches gibt, wird einfach eine Null eingetragen. Nach diesen beiden Langworten (die beiden Befehle sind zusammen ein Langwort groß) folgt die DiskFontHeader-Struktur, die so aussieht:

Die DiskFontHeader-Struktur:

```
000   ds.b   dfh_DF,14          ; Ein Exec-Node (siehe unten)
014   dc.w   dfh_FileID         ; $0f80
016   dc.w   dfh_Revision       ; Revisionsnummer des Fonts
018   dc.l   dfh_Segment        ; Segment-Adresse nach Laden
022   ds.b   dfh_Name,256       ; Name der Fontdatei
278   ds.b   dfh_TF,52          ; Eine TextFont-Struktur
330          dfh_SIZEOF
```

dfh_DF
Diese Struktur, die in die dfh-Struktur eingebettet ist, stammt aus der Exec-Library. Sie dient zum Verknüpfen der einzelnen Zeichensätze im Speicher. Genauer besprochen wird sie im Exec-Kapitel.

dfh_FileID
Das Wort $0f80 kennzeichnet die Datei als Font-Datendatei.

dfh_Revision
Eine Art interne Versionsnummer des Zeichensatzes (mehr oder weniger Spielerei).

dfh_Segment
Hier wird nach dem Einladen die Startadresse der als Segment geladenen Fontdatei eingetragen.

dfh_Name
Entspricht dem FileName-Eintrag in der FontContents-Struktur.

dfh_TF
Eingebettet in die dfh-Struktur ist auch eine TextFont-Struktur, die zur Verarbeitung des Zeichensatzes unerläßlich ist. Sie enthält alle weiteren wichtigen Daten.

Kapitel 7
Die Graphics-Library

Die wichtigsten Graphics-Strukturen

Einstellen der Zeichenfarben

Einfache Zeichenroutinen

Die Area-Zeichenroutinen

Textausgabe und Zeichensätze

Kopieren und Scrollen von Grafik

Rast- und Viewports

Sprites, VSprites und Bobs

Das IFF-Grafikformat

Die Basisstruktur der Graphics-Library

Wenn wir die Intuition-Library benutzen und uns von ihr Windows, Menüs oder Gadgets darstellen lassen, brauchen wir uns um das eigentliche Zeichnen der Grafiken nicht zu kümmern. Als nächstes wollen wir im Betriebssystem eine Stufe tiefer gehen und uns ansehen, wie wir all das, was von Intuition automatisch gezeichnet wird (Linien, Blöcke, Punkte usw.), selbst auf den Bildschirm bringen können.

Dazu bedienen wir uns der Graphics-Library. Sie ist für die erwähnten einfachen Zeichenroutinen zuständig, umfaßt aber auch komplexere Dinge wie Textausgabe, Spriteverwaltung, Display-Handling und Bildschirmscrolling. Ehe wir mit einfachen Zeichenroutinen beginnen, brauchen wir noch ein paar allgemeine Grundlagen zur Benutzung der Graphics-Library.

7.1 Die wichtigsten Graphics-Strukturen

7.1.1 Die Rastport-Struktur

Ähnlich, wie wir bei den Intuition-Routinen Window-, Screen- und sonstige Strukturen verwendeten, um unsere Objekte anzusprechen, gibt es in Graphics drei Strukturen, die das anzusprechende "Grafik-Fenster" bestimmen. Die eine ist die RastPort-Struktur. Einen RastPort kann man als Graphics-Instanz eines Intuition-Fensters bezeichnen. Er enthält Informationen über die Farben, die beim Zeichnen verwendet werden sollen, über den Zeichensatz, die Cursorposition usw. Rastports können sich, genau wie Fenster, überlagern. Wenn wir in einen Rastport zeichnen, wird nur der Teil sichtbar, der nicht von anderen Rastports überlagert wird.

Für jeden Intuition-Screen und jedes Intuition-Fenster wird automatisch ein Rastport eingerichtet und in der Screen- bzw. Window-Struktur eingetragen. Wenn wir in ein Fenster oder einen Screen zeichnen wollen, brauchen wir also die Rastport-Startadresse nur aus der entsprechenden Struktur abzulesen. Es besteht aber auch die Möglichkeit, Rastports unabhängig von Intuition-Objekten einzurichten. Diese werden dann aber nicht automatisch auf dem Bildschirm dargestellt, das muß man dann schon selbst übernehmen. Wie das geht, werden wir später noch sehen.

Die RastPort-Struktur sieht folgendermaßen aus (nicht alle Einträge sind für den Programmierer interessant):

Die RastPort-Struktur

```
000   dc.l   *rp_Layer        ; Zeiger auf den Layer des Rastports
004   dc.l   *rp_BitMap       ; Zeiger auf zugehörige BitMap
008   dc.l   *rp_AreaPtrn     ; Zeiger auf Füllmuster
012   dc.l   *rp_TmpRas       ; Zeiger auf Zwischenspeicher
```

```
016    dc.l   *rp_AreaInfo        ; Zeiger auf Area-Struktur
020    dc.l   *rp_GelsInfo        ; Zeiger auf GelsInfo-Struktur
024    dc.b    rp_Mask
025    dc.b    rp_FgPen           ; Vordergrundfarbe
026    dc.b    rp_BgPen           ; Hintergrundfarbe
027    dc.b    rp_AOLPen          ; Begrenzungsfarbe beim Füllen
028    dc.b    rp_DrawMode        ; Zeichenmodus
029    dc.b    rp_AreaPtSz        ; Anzahl Worte im Füllmuster
030    dc.b    rp_Dummy           ; Nicht benutzt
031    dc.b    rp_linpatcnt
032    dc.w    rp_Flags           ; System-Flags
034    dc.w    rp_LinePtrn        ; Linienmuster
036    dc.w    rp_cp_x            ; x-Position des Zeichencursors
038    dc.w    rp_cp_y            ; y-Position des Zeichencursors
040    ds.b    rp_minterms,8      ; Blittermaske (Grafik-Hardware)
048    dc.w    rp_PenWidth        ; Breite des Zeichenstiftes
050    dc.w    rp_PenHeight       ; Höhe des Zeichenstiftes
052    dc.l   *rp_Font            ; Zeiger auf TextFont-Struktur
056    dc.b    rp_AlgoStyle       ; Font-Stil
057    dc.b    rp_TxFlags         ; Font-Flags
058    dc.w    rp_TxHeight        ; Höhe des Fonts
060    dc.w    rp_TxWidth         ; Breite des Fonts
062    dc.w    rp_TxBaseline      ; Position der Font-Grundlinie
064    dc.w    rp_TxSpacing       ; Leer-Abstand der Fontzeichen
066    dc.l   *rp_RP_User         ; Zeiger auf Reply-Port
070    ds.b    rp_reserved,30     ; Reserviert für Erweiterungen
100            rp_SIZEOF
```

***rp_Layer**
Layers werden vom System zur Verwaltung der Zeichenebenen eingesetzt, damit entsprechend reagiert werden kann, wenn sich RastPorts überlagern. Für den Assembler-Programmierer sind sie weniger interessant.

***rp_BitMap**
In dieser Struktur ist verzeichnet, wie breit und wie hoch die Zeichenebene ist und wieviele Farben (Bitplanes) sie hat. Zu ihr werden wir später noch kommen.

***rp_AreaPtrn**
Falls Sie beim Ausfüllen einer Fläche nicht das Standard-Füllmuster (vollständig ausgefüllt) benutzen möchten, können Sie hier einen Zeiger auf ihre eigenen Füllmusterdaten eintragen.

***rp_TmpRas**
In der TmpRas-Struktur, auf die hier ein Zeiger steht, ist die Adresse eines Zwischenspeichers eingetragen, der für bestimmte Zeichenroutinen benötigt wird.

***rp_AreaInfo**
Die sog. Area-Zeichenroutinen benötigen hier einen Zeiger auf ihre Info-Struktur.

***rp_GelsInfo**
Falls Graphic-Elements (VSprites, Bobs) im Rastport angezeigt werden sollen, muß hier ein Zeiger auf ihre Info-Struktur stehen.

rp_FgPen, rp_BgPen, rp_AOLPen
Die Farbregister-Nummern für die Vordergrund-, Hintergrund- und Füllrandfarbe.

rp_DrawMode
Hier wird der derzeitige Zeichenmodus (JAM1, JAM2, COMPLEMENT oder INVERSVID) eingetragen. Genauere Beschreibung siehe Zeichenmodus-Abschnitt (7.2.3).

rp_AreaPtSz
Gibt an, wieviele Punkte das durch rp_AreaPtrn festgelegte Füllmuster hoch ist.

rp_LinePtrn
Bitmuster, das angibt, mit welchem Muster Linien gezeichnet werden sollen. Gesetzte Bits erzeugen einen Punkt in der Linie. Normalerweise steht hier $FFFF (alle Bits gesetzt), was eine durchgezogene Linie erzeugt.

rp_cp_x, rp_cp_y
Die aktuelle Position des Grafik-Cursors (pixel-genau zu setzen).

rp_PenWidth bis rp_TxSpacing
Hier finden sich die entsprechenden Einträge aus der Text-Font-Struktur des derzeit eingestellten Zeichensatzes.

***rp_RP_User**
Ein Zeiger auf einen Exec-Messageport. Dieser dient zum Senden und Empfangen von Nachrichten (genauere Beschreibung im Exec-Kapitel).

Wie gesagt werden einige Einträge dieser Struktur nur vom System benutzt. Alle brauchbaren Einträge werden später noch ausführlich beschrieben.

Vorausgesetzt, wir wollen keinen eigenen RastPort einrichten, können wir in Windows zeichnen, indem wir den Eintrag wd_RPort (Offset 50) der Window-Struktur auslesen (dort steht der RastPort-Zeiger):

```
move.l   window,a0       ; Window-Struktur-Beginn nach a0
move.l   50(a0),a1       ; Rastport-Beginn dann in a1
```

Wir können auch direkt auf den Screen zeichnen, wobei der Rastport bei Screen+86 steht. Wichtig ist, daß in der Screen-Struktur kein Zeiger auf den Rastport steht, sondern der Rastport direkt in der Screen-Struktur enthalten ist:

```
move.l   screen,a0        ; Screen-Struktur-Beginn nach a0
lea      86(a0),a1        ; Rastport-Beginn dann in a1
```

7.1.2 Die ViewPorts

Die zweite wichtige Struktur ist die ViewPort-Struktur. Sie enthält Informationen über die verwendeten Farben, über die Bildschirmmodi und zur Ansteuerung der Grafik-Hardware. Die einzelnen Einträge des ViewPort und die dritte Struktur, genannt "View-Struktur", sind bis auf den ColorMap-Eintrag im ViewPort nur dann wichtig, wenn man eigene Rast- und ViewPorts einrichten will. Die kompletten Strukturen stellen wir später vor. Zunächst setzen wir voraus, daß View und ViewPort schon eingerichtet sind, sprich ein Intuition-Screen oder -Window geöffnet ist.

In diesem Fall muß man wissen, wo man sich den Zeiger auf die ViewPort-Struktur besorgen kann. Einen ViewPort gibt es, im Gegensatz zum RastPort, nur für Screens, da die Farben und Auflösung für alle Objekte auf einem Screen gleich sind. Benötigt wird der ViewPort-Zeiger hauptsächlich für Farbänderungen. Man findet ihn in der Screen-Struktur bei Offset 44. Es gilt dasselbe wie für den Rastport im Screen: Es handelt sich nicht um einen Zeiger, sondern der Viewport ist in der Screen-Struktur direkt enthalten. Durch folgende Befehle kann man sich den Zeiger besorgen:

```
move.l   screen,a0        ; Screen-Struktur-Beginn nach a0
lea      44(a0),a1        ; Viewport-Beginn dann in a1
```

Falls man den Viewport-Zeiger ausgehend von einem Window haben möchte, muß man eine Library-Routine bemühen, da in der Window-Struktur kein solcher Zeiger vorhanden ist:

ViewPortAddress	**=**	**-300 (Graphics-Library)**

***window**	**a0**	<	Zeiger auf Window, dessen Viewport ermittelt werden soll
***viewPort**	**d0**	>	Zeiger auf zugehörigen Viewport
Erklärung			Ermittelt die Adresse eines, zu einem Window gehörigen, Viewports.

7.2 Das Einstellen der Farben

Zu diesem Thema zählen drei Dinge: die Änderung der Farbpalette, also die Einstellung einer ganz neuen Farbe, das Bestimmen der Vorder- und Hintergrundfarbe und die Festlegung des Zeichenmodus. Zunächst zur Farbpalette.

7.2.1 Einstellen der Farbpalette

Die maximale Anzahl Farben, die auf einem Bildschirm darstellbar sind, ist durch die Angabe 'Depth' (Tiefe) beim Öffnen des Screens festgelegt. Depth gibt die Anzahl der Bitplanes an, die verfügbare Farbanzahl beträgt 2^{Depth}. Zur Änderung einer Farbe aus dieser Palette gibt es drei Routinen. Die erste:

SetRGB4	**=**	**-228 (Graphics-Library)**

***viewPort**	**a0**	<	Zeiger auf ViewPort
index	**d0**	<	Nummer des zu ändernden Farbregisters
r	**d1**	<	Rotanteil der neuen Farbe (0-15)
g	**d2**	<	Grünanteil
b	**d3**	<	Blauanteil

Erklärung

Nimmt die Einstellung eines Farbregisters in einem ViewPort vor. Die neue Farbe wird dann sofort auf den Bildschirm übernommen.

Hier haben wir also das erste Einsatzgebiet für den ViewPort. Ein Anwendungsbeispiel in einem Programm wird später noch folgen. Die zweite Routine zur Farbeinstellung ist folgende:

SetRGB4CM	**=**	**-630 (Graphics-Library)**

***colorMap**	**a0**	<	Zeiger auf ColorMap
color	**d0**	<	Nummer des zu ändernden Farbregisters
r	**d1**	<	Rotanteil der neuen Farbe (0-15)
g	**d2**	<	Grünanteil
b	**d3**	<	Blauanteil

Erklärung

Ändert eine Farbe in einer ColorMap. Die Änderung wird nicht sofort am Bildschirm dargestellt, sondern erst nach Aufruf der RemakeDisplay- oder MakeScreen-Routine.

Der Unterschied zu SetRGB4 ist, daß hier ein Zeiger auf eine ColorMap-Struktur erwartet wird. Dieser ist in der ViewPort-Struktur im Eintrag vp_ColorMap (Offset 4) zu finden. Außerdem wird die neue Farbe nicht sofort sichtbar, sondern erst nach Aufruf der MakeScreen- oder RemakeDisplay-Routine:

MakeScreen	=	**-378 (Intuition-Library)**

*screen	a0	<	Zeiger auf den neu zu berechnenden Screen
Erklärung			Führt eine Neuberechnung der Hardware-Kontrollstrukturen für den angegebenen Screen durch.

RemakeDisplay	=	**-384 (Intuition-Library)**

Erklärung	Ruft MakeScreen für alle Screens auf.

Beachten Sie, daß sich die beiden Routinen in der Intuition-Library befinden. Am besten benutzen Sie allerdings SetRGB4, das erspart Ihnen die Arbeit, eine dieser Routinen aufzurufen.

Die dritte Farbroutine ermöglicht es, gleich mehrere Farben auf einen Schlag einzustellen:

LoadRGB4	=	**-192 (Graphics-Library)**

*viewPort	a0	<	ViewPort, dessen Farben eingestellt werden sollen
*colors	a1	<	Zeiger auf Farb-Feld (1 Wort pro Farbe)
count	d0	<	Anzahl der einzustellenden Farben (eingestellt werden die Farben 0 bis count-1)
Erklärung			Stellt mehrere Farben eines ViewPorts, die aus einem Farbfeld gelesen werden, ein.

Im Farbfeld muß für jede einzustellende Farbe ein Wort stehen. Das Wort, welches man am günstigsten in hexadezimal angibt, hat den Aufbau $0rgb, wobei r, g und b die Farbanteile Rot, Grün und Blau darstellen. Das oberste Nibble des Wortes wird nicht benutzt. Um eine Farbe auf Rot 12, Grün 6 und Blau 4 einzustellen, verwendet man also das Wort $0c64.

Neben den Farbeinstell-Routinen gibt es noch eine, die das Auslesen eines Farbwertes aus einer ColorMap ermöglicht:

GetRGB4	=	**-582 (Graphics-Library)**

*colorMap	a0	<	Zeiger auf die auszulesende Colormap
entry	d0	<	Nummer des gewünschten Farbeintrags

color	**d0**	>	Ermittelter RGB-Farbwert
Erklärung			Liest einen Farbwert aus einer Colormap-Struktur aus.

Hier wird wieder ein Zeiger auf eine ColorMap-Struktur erwartet (ViewPort, Offset 4). Die Ergebnisfarbe ist in einem Register zusammengefaßt. Das unterste Nibble von d0 enthält dabei den Blauwert, das zweitunterste den Grünwert und das drittunterste den Rotwert (wie bei der Einstellung mit LoadRGB4). Mit einer AND-Verknüpfung kann man problemlos die einzelnen Farbanteile isolieren:

```
d0  AND  $00f   ergibt den Blauwert
d0  AND  $0f0   ergibt den Grünwert
d0  AND  $f00   ergibt den Rotwert
```

7.2.2 Wechseln der Zeichenfarben

Die Graphics-Routinen verwenden eine Vorder- und Hintergrundfarbe. Die gesetzten Punkte des auszugebenden Objektes werden in der Vordergrundfarbe dargestellt und die gelöschte in der Hintergrundfarbe. Letztere wird natürlich nicht bei allen Grafikobjekten gebraucht, so wird ein ausgefülltes Rechteck wohl kaum gelöschte Punkte enthalten.

Wenn Sie mit Füllroutinen arbeiten, wird noch eine weitere Farbe verwendet, die Füllrandfarbe. Diese gibt an, welche Farbe als Begrenzung der zu füllenden Fläche dient.

Zum Einstellen der Farben gibt es zwei Routinen:

SetAPen	=	**-342 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, dessen Zeichenfarbe geändert werden soll
pen	**d0**	<	Farbnummer für den Vordergrund
Erklärung			Stellt die Vordergrund-Zeichenfarbe eines RastPorts ein

SetBPen	=	**-348 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, dessen Zeichenfarbe geändert werden soll
pen	**d0**	<	Farbnummer für den Hintergrund
Erklärung			Stellt die Hintergrund-Zeichenfarbe eines RastPorts ein

Zur Einstellung der Füllrand-Farbe existiert leider keine Graphics-Routine, hier muß man "von Hand" auf den RastPort zugreifen: Die betreffende Farbnummer steht im Eintrag rp_AOLPen (Offset 27) und ist ein Byte groß. Folgende Befehlssequenz kann zur Einstellung dienen:

```
move.l   rp,a0          ; Rastport-Zeiger nach a0
move.b   #2,27(a0)      ; Stellt Farbe 2 als Füllrandfarbe ein
```

Ein Beispielprogramm, in dem auch diese Routinen zur Anwendung kommen, werden wir uns ansehen, wenn wir genug "Stoff" (demonstrierwürdige Routinen) beisammen haben.

7.2.3 Einstellen des Zeichenmodus

Hierbei wird eigentlich keine Farbe ein- oder umgestellt, es wird nur festgelegt, wie die gesetzten und gelöschten Punkte eines zu zeichnenden Objekts interpretiert werden. Gelöschte Punkte können z.B. vorkommen, wenn Sie Text ausgeben. Das Grafikobjekt Textzeile wird dabei als Rechteck angesehen, in das der Text genau hineinpaßt. Die Stellen im Rechteck, an denen der Text steht, sind dann die gesetzten Punkte, und die restlichen gelten als gelöscht.

Zur Einstellung des Zeichenmodus verwendet man die Routine SetDrMd. Beachten Sie, daß der eingestellte Modus für fast alle Zeichenroutinen der Graphics-Library gilt.

SetDrMd	=	**-354 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, dessen Zeichenmodus gesetzt werden soll
drawMode	**d0**	<	Gewünschter Zeichenmodus
Erklärung			Stellt den Zeichenmodus in einem Rastport ein.

Folgende Zeichenmodi, die bis auf JAM1 auch miteinander kombiniert werden können ('equ'-Werte aufaddieren), können in d0 eingetragen werden:

Zeichenmodus	Wert	Bedeutung
JAM1	0	Normal
JAM2	1	Gelöschte Punkte in Hintergrundfarbe
COMPLEMENT	2	Vorhandene Grafik NOT-verknüpfen
INVERSVID	4	Zu zeichnende Grafik invertieren

JAM1 — Der 'normale' Zeichenmodus. Gesetzte Punkte werden in der mit SetAPen gewählten Farbe gezeichnet, bei gelöschten Punkten bleibt die vorher vorhandene Grafik erhalten.

JAM2	Gesetzte Punkte werden in der SetAPen-Farbe gezeichnet, gelöschte in der SetBPen-(Hintergrund) Farbe. Die Grafik an den Stellen mit gelöschten Punkten bleibt also nicht erhalten.
COMPLEMENT	Die mit SetAPen und SetBPen eingestellten Farben haben hier keine Bedeutung. Die vorher vorhandene Grafik wird komplementiert, d.h. die Farbnummer (Palettennummer, nicht direkte Farbe!) des Punktes wird mit NOT-verknüpft. Aus einem Punkt mit der Farbe Nr. 9 (binär 1001) würde also ein Punkt mit der Farbe 6 (binär 0110), aus Farbe 5 (%101) würde 2 (%010). Von dieser Komplementierung sind alle Punkte betroffen, an deren Position sich ein gesetzter Punkt im zu zeichnenden Objekt befindet. Wird COMPLEMENT mit JAM2 kombiniert (equ-Wert 3), sind auch die Punkte betroffen, an deren Position sich ein gelöschter Objekt-Punkt befindet. Die gesetzten und gelöschten Punkte im Objekt werden dann also gleich behandelt.
INVERSVID	Eine Kombination mit diesem Wert bewirkt, daß die gesetzten und gelöschten Punkte im zu zeichnenden Objekt invertiert, also umgedreht werden. Vor allem Texte können so sehr leicht invertiert ausgegeben werden.

Eine gute Anwendungsmöglichkeit für den Zeichenmodus COMPLEMENT findet man in vielen Malprogrammen: Wenn man dort z.B. ein ausgefülltes Rechteck zeichnen will, kann man mit der Maus seine Größe bestimmen. Dabei wird bei jeder Mausbewegung ein Rechteck gezeichnet, das die gegenwärtig gewählte Größe angibt. Solche Rechtecke werden im Modus COMPLEMENT ausgegeben, da man sie dann sehr einfach wieder verschwinden lassen kann, nämlich durch nochmaliges Zeichnen in COMPLEMENT (Ausgangsgrafik zweimal NOT-verknüpft ergibt wieder die Ausgangsgrafik).

7.3 Einfache Zeichenroutinen

Zu den einfachen Zeichenroutinen zählen Punkte, Linien, Kreise, Blöcke und das Ausfüllen. Legen wir los:

7.3.1 Punkte: Zeichnen und Farbabfrage

Um einen einzelnen Punkt auf den Bildschirm zu zeichnen, benutzt man die Routine WritePixel:

WritePixel = -324 (Graphics-Library)

*rastPort	a1	<	Rastport, in dem gezeichnet werden soll
x	d0	<	x-Koordinate des Pixels
y	d1	<	y-Koordinate des Pixels
error	d0	>	-1 bedeutet, daß der Punkt außerhalb des Rastports lag.
Erklärung			Setzt einen Punkt in einem Rastport in der mit SetAPen gewählten Farbe

Wenn wir wissen wollen, welche Farbe ein bestimmter Bildpunkt hat, verwenden wir die Routine ReadPixel:

ReadPixel = -318 (Graphics-Library)

*rastPort	a1	<	Auszulesender Rastport
x	d0	<	x-Koordinate des zu lesenden Punktes
y	d1	<	y-Koordinate dieses Punktes
pen	d0	>	Nummer der Farbe in der Palette, die dieser Punkt hat oder -1, wenn der Punkt außerhalb des Rastports lag
Erklärung			Ermittelt die Farbnummer eines bestimmten Punktes in einem Rastport.

Wichtig ist, daß diese Routine nicht den direkten Farbwert des Punktes liefert, sondern die Palettennummer, mit der der Punkt gezeichnet wurde.

Nun haben wir viel gelernt, was wir in einem Komplettprogramm demonstrieren können. Sie finden es im Verzeichnis "KAPITEL_7" unter dem Namen "PRG_7_1.S". Komplett abdrucken werden wir es nicht (wegen der Länge und weil es viele Teile enthält, die in früheren Kapiteln beschrieben wurden).
Das gilt auch für die folgenden Programme. Das Programm öffnet ein Fenster auf der Workbench, merkt sich die eingestellten Farben, stellt dann andere Farben ein und läßt Sie mit der Maus Punkte zeichnen. Die linke Taste setzt einen Punkt (sie kann auch festgehalten werden) und die rechte schaltet zur nächsten Zeichenfarbe weiter. Beim Verlassen des Programms (auszulösen durch Klick auf das Close-Gadget) werden die gemerkten Workbench-Farben wieder eingestellt.

```
* Programm 7.1 (Auszug): Demonstration GetRGB4, LoadRGB4, WritePixel,
*                        SetAPen

        ...

* Diverse Zeiger besorgen

        move.l  d0,a0           ; Window nach a0
        move.l  $2e(a0),a0      ; Zeiger auf Screen
        lea     $2c(a0),a0      ; Jetzt zum ViewPort
        move.l  a0,a5           ; ViewPort merken
        move.l  4(a0),a4        ; Zeiger auf ColorMap merken

* Alte Workbench-Farben merken

        moveq   #0,d4           ; Zähler für Farben
        lea     oldcol,a3       ; Zeiger auf Feld für alte Farben
main2:  move.l  a4,a0           ; ColorMap nach a0
        move.l  d4,d0           ; Farbnummer
        jsr     GetRGB4(a6)
        move.w  d4,d1
        asl.w   #1,d1
        move.w  d0,0(a3,d1.w)   ; Farbe in Farbfeld eintragen
        addq    #1,d4
        cmp.b   #4,d4
        blt     main2

* Neue Farben einstellen

        move.l  a5,a0           ; ViewPort-Zeiger nach a1
        lea     colors,a1       ; Zeiger auf Farbfeld nach a1
        moveq   #4,d0           ; Setze 4 Farben
        jsr     LoadRGB4(a6)    ; LoadRGB4 aufrufen

        ...

mbutton:
        cmp.l   #$68,d5         ; Linke Taste gedrückt?
        beq     mb1             ; Wenn ja
        cmp.l   #$e8,d5         ; Linke Taste losgelassen?
        beq     mb2             ; Wenn ja
        cmp.l   #$69,d5         ; Rechte Taste gedrückt?
        bne     mb3             ; Wenn nein

* Farbnummer erhöhen

        add.b   #1,drawcol      ; Nächste Farbe anwählen
        cmp.b   #3,drawcol      ; Höchste Farbnummer überschritten?
        ble     mb3             ; Wenn nein
        move.b  #0,drawcol      ; Zur Farbe 0 zurück
        bra     mb3

mb1:    move.b  #1,drawon       ; Zeichnen aktivieren
        bra     mmove
```

```
mb2:    clr.b   drawon          ; Zeichnen deaktivieren

mb3:    bra     main1           ; Zur Hauptschleife

mmove:  tst.b   drawon          ; Zeichnen aktiviert?
        beq     mm1             ; Wenn nein

* Punkt an Mauskoordinaten setzen

        move.l  gfxbase,a6
        move.l  a4,a1           ; Rastport nach a1
        move.b  drawcol,d0      ; Farbe nach d0
        jsr     SetAPen(a6)     ; Farbe setzen

        move.l  a4,a1           ; Rastport
        move.w  d6,d0           ; Maus-x-Position
        move.w  d7,d1           ; Maus-y-Position
        jsr     WritePixel(a6); Punkt zeichnen

mm1:    bra     main1           ; Zur Hauptschleife

        ...
```

Programm 7.1 (Auszug)

Zunächst besorgen wir uns diverse Zeiger. In der Window-Struktur (Zeiger in a0) steht ab Offset $2e der Zeiger auf den Screen, auf dem das Fenster liegt. In dessen Struktur ist ab $2c ein ViewPort eingebettet (deshalb LEA und nicht MOVE.L), und in diesem ab Offset 4 der ColorMap-Zeiger.

Nun müssen wir uns die derzeitig eingestellten Workbench-Farben merken. d4 wird unser Farbnummer-Zähler, in a3 kommt die Startadresse des vier Worte großen Puffers. Dann rufen wir für jede der vier Farben GetRGB4 auf und schreiben das Ergebnis nach a3 (Pufferstart) plus d4 (Farbnummer) mal 2 (ein Wort oder zwei Bytes pro Farbe).

Nachdem wir mit LoadRGB4 unsere Farben eingestellt haben, können wir auf Nachrichten vom Fenster warten. Wenn eine Meldung eintrifft, merken wir uns die wichtigen Werte aus der IntuiMessage (Class, Code, MouseX und MouseY). Als Nachricht lassen wir CLOSEWINDOW (Sprung zum Programmende), MOUSEBUTTONS und MOUSEMOVE zu.

In der MOUSEBUTTONS-Routine wird geprüft, was mit welcher Maustaste angestellt wurde. Bei Drücken der linken Taste wird ein Flag gesetzt, daß ab jetzt gezeichnet werden soll. Bei Loslassen der linken Taste wird selbiges gelöscht. Bei Druck auf die rechte Taste wird die Zeichenfarbnummer um eins erhöht und auf 0 gesetzt, wenn die 3 überschritten wurde (die Workbench hat die Farben 0-3). Im Falle von linke Taste gedrückt wird zur Zeichenroutine gesprungen, ansonsten zur Hauptschleife.

Die Zeichenroutine, die immer bei Verwendung von MOUSEMOVE aufgerufen wird, ist ganz einfach. Wenn das Zeichen-Flag gesetzt ist, wird die Zeichenfarbe mit SetAPen eingestellt und an die Maus-Koordinaten (gemerkt zum Zeitpunkt des Eintreffens der MOUSEMOVE-Nachricht) mit WritePixel ein Punkt gesetzt.

In der Finish-Routine werden vor dem Schließen des Fensters usw. die alten Farben per LoadRGB4 wieder eingestellt.

Wenn Sie das Programm ausprobieren, wird Ihnen bestimmt auffallen, daß bei schnellen Mausbewegungen keine durchgezogenen Linien gezeichnet werden. Das liegt daran, daß Intuition die Mausposition nur in gewissen Intervallen abfragen kann und deshalb bei hoher Geschwindigkeit einige Positionen "verpaßt". Im nächsten Beispielprogramm werden wir dieses Manko ausgleichen.

7.3.2 Setzen des Stiftes und Zeichnen von Linien

In jedem Rastport gibt es einen sog. "Zeichenstift", der auf einen Punkt zeigt und daher auch punkt-genau zu setzen ist. Seine derzeitige Position ist in der RastPort-Struktur vermerkt. Diesen Stift kann man über den Rastport bewegen, und zwar entweder mit oder ohne gleichzeitigem Zeichnen. Die Bewegung ohne Zeichnen besorgt die Move-Routine:

Move	=	**-240 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, dessen Stiftposition geändert werden soll
x	**d0**	<	Neue x-Position
y	**d1**	<	Neue y-Position
Erklärung			Versetzt den Zeichenstift in einem Rast-Port, ohne dabei zu zeichnen.

Das Verschieben mit Zeichnen besorgt die Draw-Routine:

Draw	=	**-246 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, dessen Stiftposition geändert werden soll
x	**d0**	<	Neue x-Position
y	**d1**	<	Neue y-Position
Erklärung			Versetzt den Zeichenstift in einem Rast-Port und zeichnet dabei eine Linie in der APen-Farbe mit dem LinePtrn-Muster von der alten Stiftposition zur neuen.

Nun ist klar, wie man eine Linie zeichnet: Man setzt den Zeichenstift mit Move auf den Startpunkt und zieht ihn mit Draw zum Endpunkt. Sie haben vielleicht erwartet, daß es nur eine Graphics-Linienroutine gibt, der man Start- und Endpunkt übergeben muß. Die verwendete Methode mit den zwei Routinen bietet aber Vorteile, wenn man zusammenhängende Linien zeichnet, also der Endpunkt einer Linie gleichzeitig der Startpunkt der nächsten ist. Dann braucht man jeweils für jede Linie nur den nächsten Endpunkt anzugeben - der Startpunkt ist ja durch den Zeichenstift, der beim Zeichnen der letzten Linie mit verschoben wurde, festgelegt.

Außerdem gibt es noch einige weitere Routinen, die sich auf die mit Move (oder Draw) gewählte Zeichenstiftposition beziehen, z.B. Text oder ClearEOL.

Mit unserem jetzigen Wissen können wir das letzte Beispielprogramm noch etwas aufpeppen: Es hatte ja den Nachteil, daß aufgrund der intervallartigen Mausabfrage durch Intuition bei schnellen Mausbewegungen keine durchgezogenen Linien gezeichnet wurden. Das werden wir nun ändern. Das zugehörige Programm finden Sie unter dem Namen "PRG_7_2.S" auf der Diskette.

```
* Programm 7.2 (Auszug): Demonstration GetRGB4, LoadRGB4, SetAPen,
                         WritePixel, Move, Draw

        ...

        move.l  a4,a1           ; Rastport nach a1
        move.b  drawcol,d0      ; Farbe nach d0
        jsr     SetAPen(a6)     ; Farbe setzen

        move.l  a4,a1           ; Rastport
        move.w  d6,d0           ; Maus-x-Position
        move.w  d7,d1           ; Maus-y-Position
        jsr     WritePixel(a6); Punkt zeichnen

        move.l  a4,a1           ; Rastport
        move.w  d6,d0           ; x-Pos
        move.w  d7,d1           ; y-Pos
        jsr     Move(a6)        ; Zeichenstift repositionieren

        ...

        move.l  a4,a1           ; Rastport
        move.w  d6,d0           ; Maus-x-Position
        move.w  d7,d1           ; Maus-y-Position
        jsr     Draw(a6)        ; Linie zum neuen Punkt ziehen
```

Programm 7.2 (Auszug)

Viel hat sich nicht geändert. In der Routine, die das Drükken der linken Maustaste bearbeitet, wird neben dem Setzen des Zeichen-Flags an die angeklickte Position ein Punkt gesetzt und außerdem der Zeichenstift mit Move dorthin gebracht. In der Routine für Mausbewegungen wurde lediglich der WritePixel-Aufruf durch einen Draw-Aufruf ersetzt.

Ergebnis: Wird die Maustaste nur gedrückt und wieder losgelassen, wird an dieser Stelle ein Punkt gesetzt (mit WritePixel in der mbutton-Routine). Wird bei gedrückter Taste die Maus bewegt, werden Linien gezeichnet, wodurch die Intervall-Tastenabfrage von Intuition "überlistet" wird und durchgezogene Linien erscheinen.

Zeichnen von mehreren Linien mit PolyDraw

Für den Fall, daß Sie mehrere Linien hintereinander zeichnen wollen, brauchen Sie nicht für jede die Draw-Routine aufzurufen. Es gibt eine Routine, die man als "Abkürzung" einsetzen kann, und zwar PolyDraw:

PolyDraw	=	**-336 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, in dem gezeichnet werden soll
***polyTable**	**a0**	<	Zeiger auf die Tabelle, die die Linien-Koordinaten enthält
count	**d0**	<	Anzahl der zu zeichnenden Linien
Erklärung			Zeichnet mehrere Linien, deren Koordinaten aus einer Tabelle gelesen werden, in einen Rastport (Abkürzung für mehrmaligen Aufruf von Draw).

In der Tabelle werden für jede Linie die Koordinaten des Zielpunktes angegeben. Die x- und y-Koordinaten sind jeweils ein Wort groß. Da der Zeichenstift beim Linienzeichnen mitbewegt wird, ist der Endpunkt jeder Linie gleichzeitig der Anfangspunkt der nächsten (genau wie bei Draw). Ein Beispiel für einen PolyDraw-Aufruf, der ein Quadrat auf den Bildschirm zeichnet:

```
        move.l  a4,a1           ; Rastport stehe in a4
        move.l  #100,d0         ; Das Quadrat soll bei 100/100 beginnen,
        move.l  #100,d1         ; dazu muß der Stift dorthin geMovet
        jsr     -240(a6)        ; werden, da PolyDraw wie Draw arbeitet

        lea     poly,a0         ; Zeiger auf Punkt-Tabelle
        move.l  #5,d0           ; Zeichne 4 Linien
        jsr     -336(a6)        ; PolyDraw anspringen
        ...
```

```
poly:       dc.w    150,100         ; Zeichne nach 150/100
            dc.w    150,150         ; Dann nach 150/150
            dc.w    150,100         ; usw.
            dc.w    100,100
```

Bild 7.1: Zeichnen von mehreren Linien mit PolyDraw

Änderung des Linienmusters

Nun zum Linienmuster. In der Rastport-Struktur gibt es ab Offset 34 einen Wort-Eintrag namens rp_LinePtrn. Dieses Wort dient in Binärdarstellung als Pixel-Muster, in dem alle Linien gezeichnet werden. Ein gesetztes Bit im Muster erzeugt einen gesetzten Punkt und ein gelöschtes Bit einen gelöschten Punkt. Falls die zu zeichnende Linie länger als 16 Pixel (so viele Bits hat ein Wort) ist, wird im Muster wieder von vorne begonnen. Der Standardwert für das Muster ist $FFFF (alle Bits gesetzt, durchgezogene Linie).

Das Muster kann jederzeit durch Zugriff auf den Rastport geändert werden. Um ein unterbrochenes Linienmuster anzuwählen, könnte folgender Befehl dienen:

```
        move.w  #%1010101010101010,34(a1)   oder
        move.w  #$aaaa,34(a1)
```

wenn der Rastport-Zeiger in a1 steht.

7.3.3 Zeichnen von Kreisen und Ellipsen

Eine Ellipse ist im Prinzip ein Kreis mit getrennt anzugebenden Radien für die horizontale und vertikale Richtung. Der Amiga geht den "umgekehrten" Weg: Er stellt nur eine Routine zum Zeichnen von Ellipsen zur Verfügung; einen Kreis erstellt man einfach als Ellipse mit identischen Radien. Die zuständige Routine heißt DrawEllipse:

DrawEllipse	**=**	**-180 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, in dem gezeichnet werden soll
cx	**d0**	<	x-Koordinate des Mittelpunkts
cy	**d1**	<	y-Koordinate des Mittelpunkts
a	**d2**	<	Horizontaler Radius
b	**d3**	<	Vertikaler Radius

Erklärung Zeichnet eine Ellipse in der APen-Farbe in einen Rastport

Wichtig ist, daß Sie die Auflösung des Screens beachten, auf den Sie zeichnen wollen. Beispiel: Bei einem Screen mit horizontal hoher und vertikal niedriger Auflösung wirkt sich

der Vertikalradius doppelt so stark aus wie der Horizontalradius, da die Pixel doppelt so hoch wie breit sind. Wenn Sie die Radien gleich einstellen, wird die Ellipse also doppelt so hoch wie breit sein. Um Kreise auf solchen Screens zu zeichnen, muß der Horizontalradius immer doppelt so groß sein wie der Vertikalradius. Bei Screens mit anderen Auflösungen gilt Entsprechendes. Falls die Auflösungen des Screens horizontal und vertikal gleich sind, können Sie die Radien "ganz normal" wählen.

Zum Thema Ellipsen wollen wir uns auch ein Demoprogramm anschauen. Es zeichnet eine Anzahl von konzentrischen Ellipsen mit variierenden Vertikal- und Horizontalradien in einen Screen mit vertikal und horizontal niedriger Auflösung, also einem LORES-Screen (damit es keinen Ärger mit den Radien gibt). Das Programm steht unter "PRG_7_3.S" auf der Diskette.

Der Kern des Programms steckt in einer Schleife, in der zunächst eine Ellipse mit festem Mittelpunkt und variablen Radien ausgegeben wird:

```
main1:  move.l   a4,a1           ; Rastport nach a1
        move.l   #160,d0         ; Mittelpunkt: Bildschirmmitte
        move.l   #125,d1
        move.l   xrad,d2         ; Radien nach d2 und d3
        move.l   yrad,d3
        jsr      DrawEllipse(a6)
```

Die Variablen xrad und yrad finden Sie im Datenbereich:

```
xrad:       dc.l    140          ; Merker für x-Radius
yrad:       dc.l    0            ; Merker für y-Radius
```

Die Ellipse startet also mit einem x-Radius von 140 und einem y-Radius von 0. Das entspricht einer waagerechten Linie, die in der Mitte des Bildschirms liegt. Bei jedem Schleifendurchlauf werden nun die Radien verändert:

```
        sub.l    #7,xrad         ; x-Radius plus 7
        add.l    #5,yrad         ; y-Radius minus 5
        cmp.l    #120,yrad       ; y-Radius größer als 120?
        bgt      finish          ; Wenn ja, Ende
        cmp.l    #0,xrad         ; x-Radius unter Null?
        blt      finish          ; Wenn ja
        bra      main1           ; Nächste Ellipse
```

Der x-Radius wird jeweils um 7 kleiner und der y-Radius um 5 größer. Die Ellipse wird also immer höher und immer schmäler. Beendet wird die Schleife, wenn entweder der y-Radius 120 überschreitet oder der x-Radius kleiner als 0 wird (ein DrawEllipse-Aufruf mit negativen Radien gibt nämlich einen Absturz). Das Ergebnis ist eine interessante Figur, die Sie sich am besten im Programm selbst anschauen. Sie können mit den Radien-Werten und den sub- und add-Befehlen nach Herzenslust experimentieren.

7.3.4 Normale und ausgefüllte Rechtecke

Zum Zeichnen eines nicht-ausgefüllten Rechtecks gibt es keine besondere Routine, Sie müssen dies mit einem Move- und vier Draw-Befehlen erledigen. Beispiel: Um ein Rechteck zu zeichnen, dessen Ecken bei den Koordinaten 50,50 (linke obere Ecke) und 100,100 (rechte untere Ecke) liegen, gehen Sie folgendermaßen vor:

- Move nach 50,50
- Draw nach 50,100
- Draw nach 100,100
- Draw nach 100,50
- Draw nach 50,50

Sie fahren also die vier Seiten des Rechtecks ab. Zur Erstellung eines ausgefüllten Rechtecks gibt es jedoch eine Graphics-Routine:

RectFill	=	**-306 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, in dem gezeichnet werden soll
xl	**d0**	<	x-Koord. der linken oberen Ecke
yl	**d1**	<	y-Koord. der linken oberen Ecke
xu	**d2**	<	x-Koord. der rechten unteren Ecke
yu	**d3**	<	y-Koord. der rechten unteren Ecke

Erklärung

Zeichnet ein Rechteck im angegebenen Rastport in der APen-Farbe und füllt es in der gleichen Farbe aus.

Beachten Sie, daß die End-Koordinaten (rechte untere Ecke) des Rechtecks nicht oberhalb bzw. links von den Startkoordinaten liegen dürfen (d2 bzw. d3 also kleiner sind als d0 bzw. d1). In diesem Fall gibt es nämlich einen (meistens grafisch recht intensiven) Absturz.

Das Rechteck-Beispielprogramm dient gleichzeitig auch der Demonstration des COMPLEMENT-Zeichenmodus. Es öffnet ein Window auf der Workbench und läßt Sie mit der Maus ausgefüllte Rechtecke zeichnen. Sie müssen zuerst die linke obere Ecke anfahren, dann die linke Taste festhalten, zur rechten unteren Ecke fahren und die Taste loslassen. Während der Mausbewegung bei gedrückter Taste werden Sie sehen, daß jeweils die Momentangröße des Rechtecks gezeigt wird, und zwar durch Komplementierung der schon vorhandenen Grafik. Dadurch kann das "Derzeit-Rechteck" leicht wieder entfernt werden, nämlich einfach durch nochmaliges Zeichnen (siehe auch Abschnitt "Einstellen des Zeichenmodus", 7.2.3). Sie finden das Programm unter "PRG_7_4.S" auf der Diskette.

```
* Programm 7.4 (Auszug): Demonstration RectFill und Zeichenmodus
                         COMPLEMENT

        ...

mb1:    move.l   gfxbase,a6

        move.b   #1,recton       ; Zeichnen aktivieren
        move.w   d6,xs
        move.w   d7,ys
        move.w   d6,xe
        move.w   d7,ye

        move.l   a4,a1
        moveq    #2,d0
        jsr      SetDrMd(a6)

        move.l   a4,a1
        move.w   xs,d0
        move.w   ys,d1
        jsr      WritePixel(a6)

        bra      main1

mb2:    move.l   gfxbase,a6
        clr.b    recton          ; Zeichnen deaktivieren

        move.l   a4,a1
        clr.l    d0
        jsr      SetDrMd(a6)

        move.l   a4,a1
        move.b   rectcol,d0
        jsr      SetAPen(a6)

        bsr      drawrect

        ...

mmove:  tst.b    recton          ; Zeichnen aktiviert?
        beq      main1           ; Wenn nein

        move.l   gfxbase,a6

        bsr      drawrect

        move.w   d6,xe
        move.w   d7,ye

        bsr      drawrect

        bra      main1           ; Zur Hauptschleife

        ...
```

```
drawrect:          move.l      a4,a1
        move.w  xs,d0
        move.w  ys,d1
        move.w  xe,d2
        move.w  ye,d3
        cmp.w   d0,d2
        bge     dr1
        exg     d0,d2
dr1:    cmp.w   d1,d3
        bge     dr2
        exg     d1,d3
dr2:    jsr     RectFill(a6)
        rts
```

Programm 7.4 (Auszug)

Das Programm arbeitet wieder mit Auswertung der Messages MOUSEBUTTONS und MOUSEMOVE. In der Routine 'linke Taste gedrückt' wird eine Marke gesetzt, die anzeigt, daß ab jetzt die "Derzeit-Rechtecke" gezeichnet werden sollen. In die Variablen xs, ys, xe und ye, welche die Start- und Endkoordinaten des Rechtecks beinhalten, werden die derzeitigen Mauskoordinaten geschrieben. Anschließend wird der Zeichenmodus COMPLEMENT eingeschaltet und an die Mausposition ein Punkt gesetzt.

In der Routine 'linke Taste losgelassen' wird das Rechteck-Flag gelöscht, der Zeichenmodus wieder auf JAM1 zurückgestellt, die Farbe gemäß dem Inhalt der Variablen "rectcol" eingestellt und die Rechteck-Zeichen-Unterroutine "drawrect" aufgerufen.

Beim Bewegen der Maus wird zuerst das vorige Rechteck gelöscht, indem es in COMPLEMENT noch einmal an die gleiche Stelle gezeichnet wird. Danach ist wieder die Grafik, die vor dem Zeichnen des Rechtecks da war, zu sehen. Dann wird die Rechteck-Endkoordinate in den Variablen xe und ye auf den aktuellen Mausstand gesetzt und das Rechteck erneut gezeichnet.

Die Rechteckzeichen-Unterroutine schließlich versorgt die Register mit den für RectFill notwendigen Daten. Falls die Endkoordinate für x oder y kleiner sein sollte als die Startkoordinate (was ja nicht sein darf), werden Start und Ende entsprechend vertauscht. Das dient dazu, daß man die Rechtecke in alle Richtungen zeichnen kann.

Zeichenprogramme bieten gewöhnlich die Funktion des Linienzeichnens, wobei der Start- und Endpunkt auf die gleiche Weise mit der Maus bestimmt wird, wie beim Zeichnen von Rechtecken. Unsere Methode mit dem COMPLEMENT-Zeichnen können wir auch problemlos auf Linien übertragen. Im Programm "PRG_7_5.S" ist dies geschehen.

```
* Programm 7.5 (Auszug): Demonstration Move, Draw und Zeichenmodus
                         COMPLEMENT

        ...

drawline:
        move.l  a4,a1
        move.w  xs,d0
        move.w  ys,d1
        jsr     Move(a6)
        move.l  a4,a1
        move.w  xe,d0
        move.w  ye,d1
        jsr     Draw(a6)
        rts

        ...
```

Programm 7.5 (Auszug)

Das einzige, was sich an diesem Programm geändert hat, ist die Haupt-Zeichenroutine (jetzt "drawline" und nicht mehr "drawrect"). Hier wird per Move der Startpunkt der Linie angesprungen und per Draw die Linie zum Endpunkt gezogen. Einfach, aber wirkungsvoll.

7.3.5 Das Ausfüllen von Flächen

Manchmal will man noch weitere ausgefüllte Figuren erstellen. Dazu muß man zuerst die Umrandung der Figur mit Draw, Ellipse o.ä. zeichnen und diese anschließend ausfüllen. Bevor wir die Routine benutzen können, müssen wir allerdings ein bißchen Vorarbeit leisten.

Die temporären Rastports

Ein temporärer (zeitweiser) Rastport ist quasi ein Grafik-Zwischenspeicher, der in bestimmten Fällen von der Füllroutine benötigt wird. Welche Fälle das sind, werden wir gleich sehen. Jetzt wollen wir zuerst erfahren, wie man diesen temporären Grafikspeicher anmelden kann. Dazu benötigen wir eine neue (sehr kurze) Struktur:

Die TmpRas-Struktur (Temporal Rastport)

```
00      dc.l    *tr_RasPtr          ; Zeiger auf den Speicherbereich
04      dc.l     tr_Size            ; Größe dieses Bereichs
08               tr_SIZEOF
```

***tr_RasPtr**
Ein Zeiger, der auf den Beginn des Speicherbereiches weist, der für den zeitweisen Rastport verwendet werden soll.

tr_Size
Die Größe des Grafikzwischenspeichers. Er muß mindestens so groß sein, daß das größte auszufüllende Objekt hineinpaßt, also im Zweifelsfalle die Größe des Bildschirms besitzen.

Diese Struktur richtet man am besten auf folgende Weise ein:

Zuerst läßt man sich vom System Speicherbereich für den temporären Rastport reservieren. Dafür stellt Graphics eine komfortable Routine zur Verfügung, der man nur die benötigte Höhe und Breite der Grafik mitteilt. Sie rechnet dann automatisch die Byte-Größe der Grafik aus und reserviert entsprechend Speicherplatz im Chip-Memory. Die Reservierung von Speicherbereich wird im Exec-Kapitel noch vertieft werden.

AllocRaster	=	**-492 (Graphics-Library)**

width	**d0**	<	Breite des Grafik-Rasters in Punkten
height	**d1**	<	Höhe des Grafik-Rasters in Punkten
***planeptr**	**d0**	>	Start des reservierten Speicherbereichs oder 0, falls nicht genügend freier Speicher verfügbar war.
Erklärung			Rechnet aus der Grafikhöhe und -breite die benötigte Speicherplatzgröße aus und reserviert diesen Speicherplatz im Chip-Memory.

Sie können den Speicherplatz für den temporären Rastport natürlich auch selbst bereitstellen (z.B. mit einem 'ds.b'-Befehl), müssen aber dafür sorgen, daß der Speicherbereich im Chip-Memory liegt, da die Graphics-Routinen ihn zum Zugriff auf die Grafikhardware benutzen, die ihrerseits nur auf das Chip-Memory zugreifen kann.

Als nächstes müssen Start und Größe des reservierten Speicherbereich in die eben vorgestellte TmpRas-Struktur eingetragen werden. Das könnte man "von Hand" (mit MOVE-Befehlen) tun, doch es gibt dafür auch eine Graphics-Routine:

InitTmpRas	=	**-468 (Graphics-Library)**

***tmpras**	**a0**	<	Zeiger auf den Speicherbereich, in dem die TmpRas-Struktur eingerichtet werden soll

***buff**	**a1**	<	Startadresse des Speicherbereichs für der temporären Rastport
size	**d0**	<	Größe des Rastport-Speicherbereichs
Erklärung			Stellt aus den Angaben in a1 und d0 eine TmpRas-Struktur ab der Adresse in a0 zusammen.

Die 8 Bytes für die TmpRas-Struktur stellt man am günstigsten mit einem 'ds.b'-Befehl im Programm bereit. Nachdem die TmpRas-Struktur eingerichtet ist, braucht man nur noch einen Zeiger auf sie in der RastPort-Struktur des Rastports, in dem ausgefüllt werden soll, zu setzen. Er muß im Langwort rp_TmpRas (Offset 12) abgelegt werden. Das waren die Vorarbeiten, nun können wir das eigentliche Füllen besprechen.

Die Flood-Routine

Die Routine, die beliebig begrenzte Flächen ausfüllt, heißt Flood (zu deutsch Flut):

Flood	**=**	**-330 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, in dem gefüllt werden soll
x	**d0**	<	x-Koordinate eines beliebigen Punktes innerhalb der zu füllenden Fläche
y	**d1**	<	y-Koordinate dieses Punktes
mode	**d2**	<	Füllmodus
Erklärung			Füllt eine beliebig begrenzte Fläche in einem Rastport mit der APen-Farbe aus

Für den Füllmodus kann eine 0 oder 1 angegeben werden. Bei einer 0 wird der Bildschirm, egal welche Farbe er hat, soweit ausgefüllt, bis Flood auf Punkte in der Begrenzungsfarbe, die im AOLPen-Eintrag des Rastports festgelegt ist (siehe Abschnitt Wechseln der Zeichenfarben, 7.2.2), trifft. Sie müssen hier aufpassen, daß die Begrenzungslinien in der AOLPen-Farbe keine Lücken aufweisen, da der Bildschirm sonst "überläuft" (die Bezeichnung Flood ist dann recht wörtlich zu nehmen).

Bei Benutzung dieses Füllmodus ist die Einrichtung eines temporären Rastports nicht unbedingt nötig. Fehlt er, so kann man den Ablauf des Füllvorgangs, der je nach Größe der zu füllenden Fläche einige Sekunden dauern kann, am Bildschirm verfolgen. Wurde aber ein TmpRas eingerichtet, geschieht die eigentliche Berechnung des Füllvorgangs "verdeckt", nämlich im temporären Rastport, der nicht auf dem Bildschirm zu sehen ist. Erst wenn das Füllen beendet

ist, wird der "richtige" Zeichenrastport aktualisiert, das Ausfüllen geschieht also quasi "auf einen Schlag".

Eine 1 als Füllmodus bewirkt, daß nur Punkte, die die gleiche Farbe haben wie der, der mit x und y im Routinenaufruf festgelegt ist, ausgefüllt werden. Jede andere Farbe als die des x/y-Punktes gilt hier als Begrenzungsfarbe. Für diesen Füllmodus ist die Einrichtung eines temporären Rastports allerdings unerläßlich.

Wenn man den temporären Rastport nicht mehr braucht, sollte man den von ihm belegten Speicher wieder freigeben, da dies nicht automatisch beim Programmende geschieht.
Zur Freigabe dient die Routine FreeRaster:

FreeRaster	=	**-498 (Graphics-Library)**

***planeptr**	**a0**	<	Zeiger auf den mit AllocRaster reservierten Speicherbereich
width	**d0**	<	Seine Breite in Punkten
height	**d1**	<	Seine Höhe in Punkten
Erklärung			Gibt mit AllocRaster reservierten Speicherbereich wieder frei.

Vor der Freigabe mit dieser Routine sollte der Eintrag rp_TmpRas in der Rastport-Struktur wieder gelöscht werden, damit nicht "aus Versehen" Fülloperationen mit nicht mehr reservierten Speicher vorgenommen werden (Guru-Gefahr).

Einstellung des Füllmusters

Die Flood-Routine füllt Flächen standardmäßig vollständig aus. Falls Sie ein eigenes Füllmuster benutzen möchten, müssen Sie auf zwei Einträge in der RastPort-Struktur zugreifen. In den Eintrag rp_AreaPtrn (Offset 8) muß ein Zeiger auf ein Datenfeld, das das Füllmuster angibt, weisen. In diesem Datenfeld steht für jede Zeile des Musters ein Wort. Sie müssen also für jede Zeile eine Folge von 16 gesetzten oder gelöschten Punkten (Bits) angeben, die sich bei größeren Füllflächen wiederholen wird. Die Anzahl der Zeilen wird im Byte-Eintrag rp_AreaPtSz (Offset 29) angegeben. Der Wert, den man dort hineinschreibt, muß dabei $2^{Zeilenzahl}$ betragen, also 0 für eine Zeile, 1 für zwei Zeilen, 2 für 4 Zeilen usw. Bei Füllflächen, die höher sind als das Füllmuster, wird sich dieses auch in der vertikalen Richtung wiederholen. Das so eingestellte Füllmuster gilt für alle Graphics-Routinen, die irgendetwas ausfüllen, so auch RectFill und die Area-Routinen (letztere kommen später). Ein kleines Beispiel: Die folgende Befehlssequenz stellt ein schraffiertes Füllmuster, das acht Zeilen hoch ist, ein.

```
        move.l  rp,a1           ; Rastport nach a1
        move.l  #ptrn,8(a1)     ; Beginn der Pattern-Daten
        move.b  #3,29(a1)       ; Das Pattern ist 2³=8 Zeilen hoch
        ...

ptrn:       dc.w    %1111000011110000
            dc.w    %0111100001111000
            dc.w    %0011110000111100
            dc.w    %0001111000011110
            dc.w    %0000111100001111
            dc.w    %1000011110000111
            dc.w    %1100001111000011
            dc.w    %1110000111100001
```

Bild 7.2: Einstellung eines Schraffur-Füllmusters

Dieses Muster findet auch im Beispielprogramm "PRG_7_6.S" Verwendung. Das Programm öffnet ein Window auf der Workbench und füllt es mit dem Schraffur-Muster aus. Auch können Sie die Anwendung von AllocRaster, InitTmpRas und FreeRaster in diesem Programm in der Praxis sehen: Mit

```
        move.l  #640,d0         ; Breite 640 Pixel
        move.l  #256,d1         ; Höhe 256 Pixel
        jsr     AllocRaster(a6); Speicher für TmpRas holen
        tst.l   d0              ; keinen bekommen?
        beq     ende            ; Wenn ja, Ende
        move.l  d0,rasmem       ; Speicher merken
```

reservieren wir genug Speicher für einen 640 Punkte breiten und 256 Punkte hohen Rastport. Wir prüfen, ob wir den Speicher erfolgreich erhalten haben, springen im Fehlerfalle (d0 steht auf 0) zum Ende und sichern ansonsten den Start des Speicherbereichs. Dann legen wir die TmpRas-Struktur an:

```
        lea     tmpras,a0       ; Zeiger auf 8 Bytes für TmpRas
        move.l  rasmem,a1       ; Adresse des Rastport-Speichers
        move.l  #20480,d0       ; Länge: 20480 Bytes
        jsr     InitTmpRas(a6) ; TmpRas-Struktur einrichten
```

Wir füllen sie mit dem Start und der Größe unseres reservierten Speicherbereichs. Die Berechnung der Byte-Anzahl funktioniert so: Man teile die Breite der Grafik durch 8, da jeder Punkt durch ein Bit dargestellt wird und jedes Byte 8 Bit hat. Dann nehme man das Divisionsergebnis mit der Höhe der Grafik mal. So berechnet auch die AllocRaster-Routine den Speicherplatzbedarf.

Den Beginn der TmpRas-Struktur müssen wir noch mit

```
        lea     tmpras,a0       ; Zeiger auf TmpRas
        move.l  a0,12(a4)       ; in RastPort eintragen
```

in den Rastport eintragen. Dann setzen wir die Zeichenfarbe (wir benutzen Farbe 1) und tragen Start und Größe des Füllmusters mit

```
        move.l   #patt,8(a4)      ; Beginn der Pattern-Daten
        move.b   #3,$1d(a4)       ; Das Pattern ist 2^3=8 Zeilen hoch
```

in den Rastport ein (in a4 haben wir den Zeiger auf ihn gesichert). Nun können wir die Flood-Routine aufrufen:

```
        move.l   a4,a1            ; Rastport
        move.l   #1,d2            ; Füll-Modus = 1
        move.l   #100,d0          ; Beliebige Koordinate im auszu-
        move.l   #100,d1          ; füllenden Fenster
        jsr      Flood(a6)        ; Füllen ausrufen
```

Die Koordinaten können für jeden beliebigen Punkt innerhalb der zu füllenden Fläche stehen. Wir benutzen den Füllmodus 1, wodurch der gesamte Innenraum des Fensters bis hin zum Rand gefüllt wird. Das geschieht daher, weil wir als Startpunkt für das Füllen einen Punkt mit der Farbe 0 gewählt haben (an den Koordinaten 100/100 steht in dem neu geöffneten Fenster sicher nichts). Flood füllt im Modus 1 alle Punkte, die die gleiche Farbe wie der Startpunkt haben, wobei alle anderen Farben als Begrenzung fungieren (in unserem Fall der andersfarbige Fensterrahmen).

Nach dem obligatorischen Warten auf den Closegadget-Klick wird der Eintrag für den temporären Rastport wieder aus der Rastport-Struktur gelöscht

```
        clr.l    12(a4)           ; TmpRas-Eintrag im Rastport löschen
```

und der vorher reservierte Speicherbereich freigegeben:

```
        move.l   rasmem,a0        ; Raster-Speicher freigeben
        move.l   #640,d0
        move.l   #256,d1
        jsr      FreeRaster(a6)
```

Da wir den Füllmodus 1 benutzt haben, mußten wir einen temporären Rastport einrichten. Hätten wir 0 benutzt, wäre es auch ohne gegangen, dann hätten wir allerdings die Füllrandfarbe im rp_AOLPen-Eintrag des Rastports richtig setzen müssen.

Einfärben eines gesamten Rastports

Neben der "normalen" Füllroutine gibt es noch eine, die einen gesamten Rastport einfärbt, ohne auf irgendwelche Begrenzungslinien zu achten. Sie füllt in einer wählbaren Farbe mit dem Standard-Muster (komplett ausgefüllt). Die Routine heißt SetRast:

SetRast	=	-234 (Graphics-Library)

***rastPort**	**a1**	<	Rastport, der komplett ausgefüllt werden soll
color	**d0**	<	Nummer des Farbregisters, mit dessen Farbe der Rastport gefüllt wird

Erklärung Füllt einen gesamten Rastport in einer wählbaren Farbe aus.

7.4 Die Area-Zeichenroutinen

Zu dieser Gruppe gehören drei Routinen: Das Versetzen des Zeichenstiftes, das Versetzen mit gleichzeitigem Zeichnen von Linien und das Zeichnen von Ellipsen. Moment, werden Sie jetzt denken, das hatten wir doch alles schon. Das stimmt - fast. Der Aufruf der drei Area-Routinen AreaMove, AreaDraw und AreaEllipse ist identisch mit dem, der drei schon bekannten Routinen, allerdings werden die Zeichenbefehle nicht sofort am Bildschirm ausgeführt, sondern in eine Art "Warteschlange" geschickt, die sich in der Amiga-Sprache AreaInfo-Struktur nennt. Dort werden alle Punkte (bzw. ihre x- und y-Koordinaten) der Objekte, die mit den Area-Routinen gezeichnet werden, abgelegt. Eine weitere Routine, AreaEnd, leitet dann das Zeichnen aller Objekte in der Warteschlange ein. Alle Objekte werden automatisch mit dem eingestellten Füllmuster ausgefüllt. Sollte ein Vieleck nicht "geschlossen" sein, d.h. ist sein letzter Punkt nicht gleich seinem ersten, wird noch eine weitere Linie vom letzten angegebenen Punkt zum ersten gezogen, damit ausgefüllt werden kann.

7.4.1 Einrichtung der AreaInfo-Struktur

Bevor man überhaupt mit Area-Routinen arbeiten kann, muß der temporäre Rastport für den Rastport, in dem man zeichnen möchte, installiert sein. AllocRaster und InitTmpRas sind also wieder gefragt. Als nächstes muß die schon erwähnte AreaInfo-Struktur initialisiert werden. Hier zunächst ihr Aufbau:

Die AreaInfo-Struktur

```
00    dc.l   *ai_VctrTbl        ; Beginn der Vektortabelle
04    dc.l   *ai_VctrPtr        ; Nächster Vektoreintrag
08    dc.l   *ai_FlagTbl        ; Beginn der Flagtabelle
12    dc.l   *ai_FlagPtr        ; Nächster Flageintrag
16    dc.w    ai_Count          ; Derzeitige Vektoreintragsnummer
18    dc.w    ai_MaxCount       ; Maximalzahl Vektoreinträge
20    dc.w    ai_FirstX         ; x-Koord. des ersten Punktes
```

```
22      dc.w    ai_FirstY          ; y-Koord. des ersten Punktes
24              ai_SIZEOF
```

***ai_VctrTbl**
Die Bezeichnung "Vektor" steht bei den Area-Routinen für ein x/y-Koordinatenpaar. In ai_VctrTbl steht ein Zeiger auf den Beginn einer Wort-Tabelle, in der die Koordinaten aller Punkte der Objekte, die mit den Area-Routinen gezeichnet werden, stehen (jeweils ein Wort für x- und y-Koordinate, also insgesamt zwei Worte pro Vektoreintrag).

***ai_VctrTbl**
Dieser Zeiger zeigt auf das nächste zu belegende Wort in der Vektortabelle. Nach jedem Area-Routinenaufruf wird er entsprechend der Koordinatenanzahl hochgezählt.

***ai_FlagTbl**
Zu jedem Koordinatenpaar wird ein Flag in einer gesonderten Tabelle aufgezeichnet. Es enthält Informationen, um welchen Typ Punkt (Startpunkt eines Vielecks, Linie in einem Vieleck oder Koordinaten für eine Ellipse) es sich handelt. Die Flags sind jeweils ein Byte groß.

***ai_FlagPtr**
Analog zu ai_VctrPtr steht hier ein Zeiger auf die Stelle, an der das nächste Flag eingetragen werden soll.

ai_Count
Dies ist ein Zähler, der bei 0 startet. Für jedes eingetragene Koordinatenpaar wird er um eins erhöht.

ai_MaxCount
Gibt den Maximalwert für ai_Count, also die maximale Anzahl von Koordinatenpaaren in der Vektortabelle an.

ai_FirstX
Hier wird, zusätzlich zur Koordinatentabelle, die x-Koordinate des ersten Punkts im letzten gezeichneten Objekt eingetragen.

ai_FirstY
Analog zu ai_FirstX steht hier die y-Koordinate.

Wem diese Struktur etwas zu kompliziert vorkommt, den können wir beruhigen: Sie brauchen sich um ihre Verwaltung in keinster Weise zu kümmern. Alles, was Sie tun müssen, ist genug Speicher für Vektor- und Flagtabelle und die Info-Struktur zu reservieren, die Routine InitArea aufrufen (welche die Info-Struktur automatisch initialisiert) und den Beginn der Info-Struktur in den Rastport eintragen.

Bevor Sie an die Speicherreservierung gehen, sollten Sie festlegen, wieviele Punkte ihre Area-Objekte maximal haben sollen. Jeder AreaMove- und AreaDraw-Befehl braucht einen Vektoreintrag und jeder AreaEllipse-Befehl zwei. Sicher-

heitshalber sollten Sie ein paar Punkte mehr vorsehen, da automatisch zusätzliche Punkte eingebaut werden, wenn Start- und Endpunkt eines AreaDraw-Vielecks nicht übereinstimmen. Dann zieht die AreaDraw-Routine automatisch eine Linie vom letzten angegebenen Punkt zum Startpunkt des Vielecks, damit es "geschlossen" ist.

Jeder Punkt (jedes Koordinatenpaar) benötigt fünf Bytes (vier für die zwei Koordinaten-Worte und eins für das Flag). Der zu reservierende Speicherbereich beträgt also '5 mal Punktanzahl' Bytes. Die Reservierung nehmen Sie am besten mit einem 'ds.b'-Befehl vor. Nachfolgend die Routine, die die Einrichtung der AreaInfo-Struktur übernimmt:

InitArea	=	**-282 (Graphics-Library)**

***areaInfo**	**a0**	<	Zeiger auf den Speicherbereich für die einzurichtende AreaInfo-Struktur
***vectorTable**	**a1**	<	Zeiger auf den Speicherbereich, in dem die Koordinatenpaare und Flags abgelegt werden sollen
vtSize	**d0**	<	Anzahl der Paare, die in die Tabelle passen sollen
Erklärung			Richtet eine AreaInfo-Struktur für die Benutzung von Area-Zeichenroutinen ein.

Nachdem die Struktur eingerichtet ist, muß nur noch ein Zeiger auf sie in den gewünschten Rastport eingetragen werden, und zwar in den Eintrag rp_AreaInfo (Offset 16).

7.4.2 AreaMove, AreaDraw, AreaEllipse und AreaEnd

Jetzt können wir endlich mit den eigentlichen Area-Routinen loslegen. Sie entsprechen exakt den "normalen" Routinen Move, Draw und Ellipse.

AreaMove	=	**-252 (Graphics-Library)**

***rastPort**	**a1**	<	Rastport, dessen AreaInfo ein Move hinzugefügt werden soll
x	**d0**	<	Neue x-Koordinate
y	**d1**	<	Neue y-Koordinate
Erklärung			Hängt an die AreaInfo-Liste eines Rastports einen Move-Befehl (Versetzung des Zeichenstiftes ohne Zeichnen) an.

AreaDraw	=	**-258 (Graphics-Library)**

*rastPort	a1	<	Rastport, dessen AreaInfo ein Draw hinzugefügt werden soll
x	d0	<	Neue x-Koordinate
y	d1	<	Neue y-Koordinate
Erklärung			Hängt an die AreaInfo-Liste eines Rastports einen Draw-Befehl (Versetzung des Zeichenstiftes mit Linienzeichnen) an.

AreaEllipse	=	**-186 (Graphics-Library)**

*rastPort	a1	<	Rastport, dessen AreaInfo eine Ellipse hinzugefügt werden soll
cx	d0	<	x-Koordinate des Mittelpunkts
cy	d1	<	y-Koordinate des Mittelpunkts
a	d2	<	Horizontaler Radius
b	d3	<	Vertikaler Radius
Erklärung			Hängt an die AreaInfo-Liste eines Rastports eine Ellipse an.

Die wichtigste und auch neue Routine kommt nun: AreaEnd sorgt dafür, daß alle seit ihrem letzten Aufruf (bzw. seit der Installierung der AreaInfo-Struktur) getätigten Area-Befehle auf dem Bildschirm ausgeführt werden.

AreaEnd	=	**-264 (Graphics-Library)**

*rastPort	a1	<	Rastport, dessen AreaInfo-Zeichenbefehle ausgeführt werden sollen
Erklärung			Zeichnet alle in der AreaInfo-Struktur eingetragenen Objekte in den Rastport und entleert die AreaInfo-Struktur.

Als Zeichenfarbe für alle Objekte wird die aktuelle APen-Farbe (mit SetAPen eingestellt) benutzt. Wichtig: Alle Objekte, einschließlich ihrer Rahmen, werden nach dem Zeichnen ausgefüllt, und zwar ebenfalls in der APen-Farbe! Die AreaInfo-Struktur wird durch den AreaEnd-Aufruf entleert, aber nicht gelöscht. Sie steht sofort für neue Area-Objekte bereit.

Das zugehörige Demonstrationsprogramm zeichnet ein regelmäßiges Sechseck und eine Ellipse unter Benutzung der Area-Befehle. Sie finden es unter "PRG_7_7.S".

7.5 Textausgabe und Zeichensätze

Im Intuition-Kapitel haben wir schon die Möglichkeit der Textausgabe über PrintIText kennengelernt. Wir mußten dazu eine Struktur (IntuiText) einrichten, in der die Farben, der Zeichenmodus, die Positionen, der Zeichensatz und schließlich ein Zeiger auf den Text selber angegeben wurden. Auf Graphics-Ebene können wir auch Texte ausgeben, dazu wird allerdings keine Struktur verwendet. Die Ausgabe erfolgt über die Graphics-Routine Text, der wir nur Start und Länge unseres Textes übergeben. Ansonsten werden die Einstellungen von SetAPen, SetBPen, SetDrMd und Move übernommen.

Text	=	**-60**	**(Graphics-Library)**

***rastPort**	**a1**	<	Zeiger auf den Rastport, in dem der Text ausgegeben werden soll
***string**	**a0**	<	Zeiger auf den Beginn des Textes im Speicher. Der Text braucht nicht mit einem Nullbyte abgeschlossen zu werden.
count	**d0**	<	Länge des Textes in Bytes
Erklärung			Gibt einen Text in den angegebenen Rastport in den APen/BPen-Farben, im SetDrMd-Zeichenmodus und an der Move-Position aus.

Vor dem Aufruf von Text müssen (bzw. sollten) also die Routinen zur Einstellung von Farbe, Position etc. aufgerufen werden. Falls mehrere Texte in der gleichen Farbe oder dem gleichen Zeichenmodus ausgegeben werden, braucht SetAPen bzw. SetDrMd natürlich nur einmal aufgerufen zu werden. Die Textposition sollte aber immer vorher mit Move gesetzt werden.

Neben Text gibt es eine Routine, die die Breite eines Textes in Pixeln bestimmt (nicht zu verwechseln mit der Anzahl der Zeichen im Text). Sie berücksichtigt dabei auch den gerade eingestellten Zeichensatz. Die Routine heißt TextLength:

TextLength	=	**-54**	**(Graphics-Library)**

***rastPort**	**a1**	<	Rastport, für den die Textlänge berechnet werden soll (wichtig, da der eingestellte Font berücksichtigt wird)
***string**	**a0**	<	Zeiger auf den Textbeginn
count	**d0**	<	Länge des Textes in Zeichen
length	**d0**	>	Berechnete Breite des Textes in Pixeln

Erklärung Berechnet die Breite eines Textes in Pixeln, wobei der eingestellte Font berücksichtigt wird.

7.5.1 Zentrierte Textausgabe

Um einen Text zentriert auszugeben, muß man seine Breite in Pixeln wissen und deren Hälfte von der Mitte des Ausgabefensters abziehen. Dies ist dann die x-Startposition des Textes. Das folgende Beispielprogramm gibt den Text in der Kommandozeile zentriert in einem neuen Fenster aus (auf Diskette unter "PRG_7_8.S").

```
* Programm 7.8 (Auszug) : Zentrierte Ausgabe der
                          Kommandozeile

        ...

* Pixel-Breite des Textes bestimmen

        move.l  a4,a1           ; Rastport
        move.l  a3,a0           ; Beginn des Textes
        move.l  d3,d0           ; Länge des Textes
        jsr     TextLength(a6)  ; Routine aufrufen
        move.l  d0,d4           ; Pixelbreite merken

* Textausgabe-Position einstellen

        move.l  a4,a1           ; Rastport
        move.l  #50,d1          ; y-Koordinate 50
        move.l  #320,d0         ; Mitte des Windows
        asr     #1,d4           ; Text-Pixelbreite durch 2
        sub.l   d4,d0           ; x-Koordinate = Mitte - Breite / 2
        jsr     Move(a6)

* Text ausgeben

        move.l  a4,a1           ; Rastport
        move.l  a3,a0           ; Text-Beginn
        move.l  d3,d0           ; Text-Länge
        jsr     Text(a6)        ; ausgeben
```

Programm 7.8 (Auszug)

Zunächst rufen wir die Routine TextLength auf und lassen uns aus der Zeichenanzahl im Text die benötigte Pixelbreite feststellen. Dann berechnen wir die x-Startkoordinate des Textes. Wir schreiben 320 nach d0 (Mitte des Windows). Dann teilen wir die Pixelbreite durch zwei und ziehen das Ergebnis von der 320 in d0 ab. Als y-Koordinate wählen wir 50. Nun rufen wir Move auf, wodurch der Zeichenstift, der für

die Textausgabe als Textcursor fungiert, gesetzt wird. Da wir die Farbe und den Zeichenmodus vorher schon eingestellt haben, können wir jetzt sofort Text aufrufen.

7.5.2 Einstellung von Zeichensätzen

Im Diskfont-Kapitel haben wir gelernt, wie man Zeichensätze, die sich im ROM oder auf der Diskette befinden, öffnet. Wir wissen auch schon, wie man sie in Verbindung mit IntuiTexten verwendet oder als Screen-Standardfont einsetzt. Nun wollen wir uns ansehen, wie man einen Rastport mit einem geladenen Font versorgen kann. Dazu dient die Routine SetFont:

SetFont	**=**	**-66 (Graphics-Library)**

***rastPort**	**a1**	<	Zeiger auf den Rastport, dessen Zeichensatz eingestellt werden soll
***textFont**	**a0**	<	Zeiger auf die TextFont-Struktur des gewünschten Zeichensatzes
Erklärung			Setzt im angegebenen Rastport den Zeichensatz, auf dessen TextFont-Struktur a0 zeigt.

Wichtig: Bei den Intuition-Strukturen mußten wir zur Benutzung eines Zeichensatzes einen Zeiger auf eine TextAttr-Struktur eintragen (die gleiche Struktur, die wir zum Aufruf von OpenDiskFont benutzten). Hier müssen wir allerdings den Zeiger auf die TextFont-Struktur, also den Rückgabewert von OpenDiskFont oder OpenFont, benutzen.

Nach dem Aufruf von SetFont werden solange alle Textausgaben im neuen Zeichensatz durchgeführt, bis ein anderer eingestellt wird.

Wenn man wissen will, welcher Zeichensatz gerade eingestellt ist, verwendet man die Routine AskFont:

AskFont	**=**	**-474 (Graphics-Library)**

***rastPort**	**a1**	<	Zeiger auf den Rastport, dessen Zeichensatz abgefragt werden soll
***textAttr**	**a0**	<	Zeiger auf 8 Bytes reservierten Speicher (z.B. per 'ds.b'-Befehl), in dem eine TextAttr-Struktur angelegt wird
Erklärung			Legt eine TextAttr-Struktur mit Informationen über den momentan in einem Rastport eingestellten Zeichensatz an.

Den Speicherplatz für die Struktur legt man am besten mit dem Befehl "ds.b 8" im Datenbereich des Programms an. Die einzelnen Einträge der TextAttr-Struktur (Zeiger auf Fontname, Höhe, Stil und Flags) können nach dem AskFont-Aufruf ausgewertet werden.

Wahl des Schriftstils

Die Graphics-Routine AskSoftStyle ermittelt, welche Schriftarten für den derzeitigen Font in einem Rastport möglich sind:

AskSoftStyle		=	**-84 (Graphics-Library)**
***rastPort**	**a1**	<	Rastport, dessen mögliche Zeichensatz-Schriftarten ermittelt werden sollen
style	**d0**	>	Mögliche Schriftarten
Erklärung			Ermittelt die möglichen Schriftarten des momentan eingestellten Zeichensatzes im angegebenen Rastport.

Die Bedeutungen der Schriftstil-Werte sind folgende:

Schriftstil	Wert	Bedeutung
FSF_NORMAL	0	Kein besonderer Stil
FSF_UNDERLINED	1	Unterstrichen
FSF_BOLD	2	Fettdruck
FSF_ITALIC	4	Kursiv-(Schräg-)Druck
FSF_EXTENDED	8	Doppelte Breite (bei normalen Fonts nicht möglich)

Im Ergebnis-Register d0 stehen die aufaddierten Werte aller möglichen Schriftarten. Wenn für einen Font UNDERLINED und ITALIC zugelassen sind, wäre der Rückgabewert also 5. Die Werte müssen Sie als Wertigkeiten der Binärstellen der Rückgabezahl sehen. Ein Wert von 5 würde bedeuten, daß Bit Nr. 1 und Nr. 3 (Zählung beginnt bei 0) gesetzt sind. AskSoftStyle gibt Werte zurück, bei denen die unbenutzten Bits 5-7 (Wertigkeiten 16-128) des Style-Bytes gesetzt sein können, was aber keine Bedeutung hat.

Das Ergebnis einer AskSoftStyle-Abfrage kann, muß aber nicht, in der folgenden Routine angewandt werden, die zur Einstellung des Schriftstils dient:

SetSoftStyle	=	**-90 (Graphics-Library)**

***rastPort** **a1** < Rastport, dessen Schriftart eingestellt werden soll
style **d0** < Gewünschter Schriftstil
enable **d1** < Zugelassene Arten (von AskSoftStyle zurückgegeben)

newstyle **d0** > Wirklich gesetzter Schriftstil (ergibt sich durch OR-Verknüpfung von style und enable)

Erklärung Setzt den gewünschten Schriftstil im angegebenen Rastport

Falls die Ausgrenzung bestimmter Arten nicht gewünscht wird (die allermeisten Fonts sind sowieso in allen Schriftarten bis auf EXTENDED darstellbar), kann 'enable' auf -1 gesetzt werden, wodurch alle Bits gesetzt und alle Schriftstile zugelassen sind.

Nun ein Programm, das einen Text in einem Fenster ausgibt (unter Benutzung der Text-Routine), und zwar im Zeichensatz Garnet 16 und im Schriftstil fett und unterstrichen. Es steht unter "PRG_7_9.S" auf der Diskette.

```
* Programm 7.9 (Auszug): Graphics-Textausgabe mit Diskfonts

        ...

        lea     tattr,a0        ; Font Garnet 16 öffnen
        jsr     OpenDiskFont(a6)
        tst.l   d0
        beq     ende2
        move.l  d0,tfont

        ...

* Font und Stil einstellen

        move.l  a4,a1           ; Rastport
        move.l  tfont,a0        ; Rückgabewert von OpenDiskFont
        jsr     SetFont(a6)     ; Font einstellen

        move.l  a4,a1           ; Mögliche Stile erfragen
        jsr     AskSoftStyle(a6)
        move.l  d0,d4           ; und in d4 sichern

        move.l  a4,a1           ; Rastport
        moveq   #3,d0           ; Gewünschter Stil
        move.l  d4,d1           ; Verknüpft mit den möglichen
```

```
        jsr      SetSoftStyle(a6)

        move.l   a4,a1          ; Koordinate 100/50 setzen
        moveq    #100,d0
        moveq    #50,d1
        jsr      Move(a6)

        move.l   a4,a1          ; Text ausgeben
        lea      ausgtext,a0
        moveq    #31,d0
        jsr      Text(a6)

        ...
```

Programm 7.9 (Auszug)

Das Öffnen eines Diskfonts ist aus Kapitel 6 schon bekannt. Als nächstes werden Vordergrundfarbe und Zeichensatz eingestellt. Von AskSoftStyle lassen wir uns die zugelassenen Schriftarten des Fonts geben und schreiben sie beim SetSoftStyle-Aufruf als 'enable'-Maske nach d1. Als Schriftstil geben wir 3 an, was unterstrichen (1) und fett (2) ergibt. Mit Move springen wir nach 100/50 und geben dort den Text aus.

Wie Sie sehen, müssen wir die Länge des auszugebenden Textes wieder abzählen. Erinnern Sie sich noch an unsere Print-Routine aus dem DOS-Kapitel? Sie nahm uns die Arbeit des Textlängen-Zählens ab. Diese Routine wollen wir nun für die Graphics-Textausgabe umschreiben.

```
gprint: move.l   a2,-(sp)       ; Register sichern

        move.l   a0,a2          ; Text-Start sichern
        clr.l    d0             ; Länge löschen
gpr1:   addq     #1,d0          ; Länge plus 1
        tst.b    (a0)+          ; Textende-Kennzeichen erreicht?
        bne      gpr1           ; Wenn nein
        subq     #1,d0          ; Letzten addq rückgängig

        move.l   a2,a0          ; Textstart zurückholen
        move.l   a4,a1          ; Rastport-Zeiger nach a1
        jsr      Text(a6)       ; Text aufrufen

        move.l   (sp)+,a2       ; Register zurück
        rts                     ; Das wars
```

Bild 7.3: Eine Print-Subroutine für Graphics-Text

Die Routine arbeitet analog zur DOS-Print-Routine, wir nennen Sie 'gprint' (Graphics-Print), um sie von der anderen unterscheiden zu können.

7.5.3 Löschen des Bildschirms

Für diesen Zweck existieren zwei Routinen. Die erste:

ClearEOL	=	**-42 (Graphics-Library)**

***rastPort** **a1** < Rastport, in dem eine Zeile gelöscht werden soll

Erklärung Löscht eine Textzeile des angegebenen Rastports ab der Zeichenstiftposition bis zum Zeilenende. Der freiwerdende Bereich wird mit der Hintergrundfarbe (SetBPen) aufgefüllt.

Die Anzahl Zeilen, die dabei gelöscht wird, entspricht der Höhe des derzeit eingestellten Zeichensatzes, damit wirklich genau eine Textzeile entfernt wird. Die zweite Routine:

ClearScreen	=	**-48 (Graphics-Library)**

***rastPort** **a1** < Rastport, der gelöscht werden soll

Erklärung Löscht einen kompletten Rastport, d.h. er wird mit der Hintergrundfarbe (SetBPen) aufgefüllt.

Dazu braucht wohl nichts mehr gesagt zu werden.

7.6 Grafik-Kopier- und Scroll-Routinen

Neben der Vielzahl an Routinen, die Grafiken auf den Bildschirm bringen, gibt es auch solche, die schon vorhandene Grafiken kopieren oder verschieben. Die Graphics-Library bedient sich zu diesem Zweck eines Coprozessors, des sogenannten "Blitters". Die Bezeichnung Blitter ist eine Zusammenfassung von "Block Image Transfer", also "Blockgrafik-Kopierer". Dies beschreibt auch gleich die Hauptaufgabe des Blitters: Die Kopie von rechteckig-organisierten Grafiken. Rechteckig-organisiert bedeutet, daß die Begrenzungslinien einer zu kopierenden Grafik immer horizontal bzw. vertikal sein müssen, also keine im Ratio abweichenden Grafiken kopieren kann.

Der Blitter ist ein Spezialprozessor, dessen einzige Fähigkeit das Kopieren von Speicherbereichen (mit dazugehörigen Verknüpfungsoperationen) ist. Diese Fähigkeit beherrscht er dafür aber sehr gut, d.h. er weist eine sehr hohe Kopierge-

schwindigkeit auf. Diese hohe Geschwindigkeit nutzen auch die Graphics-Routinen aus.

7.6.1 Kopieren auf Rastport- und Bitmap-Ebene

Die erste Routine, die vorgestellt werden soll, ist ClipBlit:

ClipBlit	=	**-552 (Graphics-Library)**

***srcrp**	**a0**	<	Zeiger auf Quell-Rastport
srcx	**d0**	<	x-Startkoordinate der Quellgrafik
srcy	**d1**	<	y-Startkoordinate der Quellgrafik
***destrp**	**a0**	<	Zeiger auf Ziel-Rastport
destx	**d2**	<	x-Koordinate der Zielgrafik
desty	**d3**	<	y-Koordinate der Zielgrafik
sizex	**d4**	<	x-Größe der zu kopierenden Grafik
sizey	**d5**	<	y-Größe der zu kopierenden Grafik
minterm	**d6**	<	Logische Verknüpfung für die Kopie

Erklärung Kopiert einen rechteckigen Grafik-Ausschnitt aus einem Rastport in einen anderen (oder auch in denselben).

Wie Sie sehen, arbeitet diese Routine auf Rastport-Ebene. Rastports werden bekanntlich für Screens und Windows eingerichtet. Die Koordinaten für die Quell- und Zielgrafik sind also relativ zur oberen, linken Ecke des Screens bzw. Windows zu sehen. Wenn Sie beispielsweise ein Window haben, das bei 50/50 beginnt, bezieht sich eine Koordinatenangabe von 10/10 beim ClipBlit-Aufruf für dieses Window auf die reale Screen-Position 60/60.

Nun wollen wir uns die logische Verknüpfung (minterm) noch etwas näher ansehen. Der Koprozessor Blitter hat die Möglichkeit, drei unabhängige Quell-Grafiken in einer festlegbaren Weise zu verknüpfen und in die Ziel-Grafik zusammenzukopieren. Die Mini-Terms (wofür 'minterm' steht) sind Boolesche Gleichungen, welche die Verknüpfungsart festlegen. Im Falle der Graphics-Blitterroutinen arbeiten wir nur mit einer Quelle und einem Ziel.

Alle Blitterroutinen sehen die Quellgrafik und die am Ziel bisher vorhandene Grafik als Ausgangsgrafiken an, verknüpfen sie und schreiben das Ergebnis in die Zielgrafik. Die Verknüpfung geht punktweise vor sich, d.h. je ein Punkt aus der Quellgrafik wird mit dem entsprechenden Punkt aus der alten Zielgrafik verknüpft und der Punkt in der neuen Zielgrafik entsprechend dem Verknüpfungsergebnis gesetzt oder gelöscht. Dabei können wir in den Parameter 'minterm' Werte schreiben, welche die Art und Weise festlegen, wie die Quell- und alte Zielgrafik verknüpft werden. Zugelassen sind die Werte 128, 64, 32 und 16. Jede dieser Zahlen steht für eine ganz

bestimmte Verknüpfungsvorschrift. Falls Sie mehrere dieser Vorschriften verwenden möchten, können Sie die entsprechenden Zahlen addieren. Hier nun die Bedeutungen der einzelnen Zahlen:

Mini-Term 128	Ein Punkt in der neuen Zielgrafik ist dann gesetzt, wenn die entsprechenden Punkte in beiden Ausgangsgrafiken gesetzt waren.
Mini-Term 64	Ein Punkt in der neuen Zielgrafik ist dann gesetzt, wenn der entsprechende Punkt in der Quellgrafik gesetzt, in der alten Zielgrafik aber gelöscht war.
Mini-Term 32	Ein Punkt in der neuen Zielgrafik ist dann gesetzt, wenn der entsprechende Punkt in der Quellgrafik gelöscht, in der alten Zielgrafik aber gesetzt war.
Mini-Term 16	Ein Punkt in der neuen Zielgrafik ist dann gesetzt, wenn die entsprechenden Punkte in beiden Ausgangsgrafiken gelöscht waren.

Da die Miniterms auch beliebig addiert werden können, ergeben sich insgesamt 16 verschiedene Möglichkeiten, eine Grafik an eine andere Stelle zu kopieren. Nachfolgend die Effekte der verschiedenen Kombinationen:

240 (128+64+32+16)	Die neue Zielgrafik wird, egal was vorher in den Ausgangsgrafiken war, komplett ausgefüllt.
224 (128+64+32)	Die neue Zielgrafik entsteht durch Zusammen-Kopieren der Quellgrafik und der alten Zielgrafik: Neuziel = Quelle OR Altziel.
208 (128+64+16)	Die neue Zielgrafik entsteht durch Zusammen-Kopieren der Quellgrafik und der invertierten alten Zielgrafik: Neuziel = Quelle OR (NOT Altziel).
192 (128+64)	Die neue Zielgrafik wird durch die Quellgrafik ersetzt, egal, was in der alten Zielgrafik war.
176 (128+32+16)	Die neue Zielgrafik entsteht durch Zusammen-Kopieren der invertierten Quellgrafik und der alten Zielgrafik: Neuziel = (NOT Quelle) OR Altziel.
160 (128+32)	Die Zielgrafik wird nicht verändert.
144 (128+16)	Die Zielgrafik enthält dort gesetzte Punkte, wo in beiden Ausgangsgrafiken die Punkte entweder gesetzt oder gelöscht waren.
112 (64+32+16)	Die neue Zielgrafik enthält dort gesetzte Punkte, wo entweder in der Quellgrafik oder in der alten Zielgrafik gesetzte Punkte oder in beiden Ausgangsgrafiken gelöschte Punkte waren. An den Stellen, an denen in beiden Ausgangsgrafiken gesetzte Punkte waren, enthält die neue Zielgrafik gelöschte Punkte.
96 (64+32)	Die neue Zielgrafik enthält dort gesetzte Punkte, wo entweder in der Quellgrafik oder in der alten Zielgrafik gesetzte Punkte waren. An den Stellen, an denen in beiden Ausgangsgrafiken

	gesetzte Punkte waren, enthält die neue Zielgrafik gelöschte Punkte: Neuziel = Quelle EOR Altziel.
80 (64+16)	Die Zielgrafik wird, unabhängig von der Quellgrafik, invertiert (Ziel NOT-verknüpft).
48 (32+16)	Die Zielgrafik wird durch die invertierte Quellgrafik ersetzt (Quelle NOT-Verknüpft).
0	Die neue Zielgrafik wird, egal was vorher in den Ausgangsgrafiken war, komplett gelöscht.

Die Gründe für die Wirkungen der einzelnen Kombinationen können Sie sich klarmachen, wenn Sie überlegen, wie die Kombinationen der verschiedenen Verknüpfungs-Vorschriften auf die Punkte der Quell- und Zielgrafik wirken. Am Beispiel 192 wollen wir das einmal durchgehen:

192 bildet sich aus den Werten 128 und 64. 128 besagt, daß der neue Zielpunkt definiert sein soll, wenn beide Ausgangspunkte vorhanden sind. Bei 64 wird der Zielpunkt aktiviert, wenn der Quellpunkt gesetzt, der alte Zielpunkt aber gelöscht war. Das ergibt eine 1:1-Kopie der Quellgrafik ohne Berücksichtigung der alten Zielgrafik. Mit den übrigen Miniterm-Kombinationen kann man es analog halten.

Die nächste Kopierroutine heißt BltBitMap:

BltBitMap = -30 (Graphics-Library)

***srcbm**	a0	<	Zeiger auf Quell-Bitmap
srcx	d0	<	x-Startkoordinate der Quellgrafik
srcy	d1	<	y-Startkoordinate der Quellgrafik
***destbm**	a0	<	Zeiger auf Ziel-Bitmap
destx	d2	<	x-Koordinate der Zielgrafik
desty	d3	<	y-Koordinate der Zielgrafik
sizex	d4	<	x-Größe der zu kopierenden Grafik
sizey	d5	<	y-Größe der zu kopierenden Grafik
minterm	d6	<	Logische Verknüpfung für die Kopie
mask	d7	<	Maske für die zu kopierenden Planes
***buffer**	a2	<	Zeiger auf Zwischenspeicher bei Überlagerung der Quell- und Zielgrafiken (sollte groß genug für eine Grafik-Zeile sein).

Erklärung Kopiert einen rechteckigen Grafik-Ausschnitt aus einer Bitmap.

Im Gegensatz zu ClipBlit erwartet BltBitMap Zeiger auf sog. Bitmap-Strukturen. Im Intuition-Kapitel haben wir schon etwas über den Aufbau einer Amiga-Grafik erfahren. Zur Erinnerung: Eine Grafik besteht aus bis zu sechs Bitplanes, die alle die gleiche Spalten- und Zeilenzahl haben. Die Farbtabellennummer eines Punktes ergibt sich als Kombinations-Binärzahl aus den "übereinanderliegenden" Punkten (Bits) al-

ler Bitplanes. Die Zusammenstellung der Bitplanes zu einer Grafik nennt man Bitmap. Sie stellt die unterste Stufe der Grafikverwaltung dar.

In jedem Rastport findet man einen Zeiger auf die Bitmap-Struktur, in der die Grafik verwaltet wird (Eintrag rp_BitMap, Offset 4). Alle Rastports (und damit alle Windows) auf einem Screen werden in ein und derselben Bitmap dargestellt. Überlagerung und Hintergrund/Vordergrund wie bei den Windows gibt es hier nicht, denn irgendwo müssen die, miteinander verknüpften Fenster, in einer einzigen Grafik zusammengefaßt werden, damit sie auf dem Bildschirm erscheinen können.

Im Zusammenhang mit der BltBitMap-Routine ist es wichtig, daß die anzugebenden Koordinaten nicht relativ zur linken oberen Ecke des Rastports, sondern relativ zur Oberkante des Screens, auf dem der Rastport liegt, zu sehen sind. Hat man also ein Window, das bei 50/50 auf dem Screen beginnt, so muß man, um die window-relativen Koordinaten 10/10 zu erreichen, die Koordinaten 60/60 an BltBitMap übergeben.

Die Benutzung von BltBitMap bietet insofern Vorteile gegenüber ClipBlit, als Sie hier angeben können, welche Planes der Bitmap kopiert werden sollen. Angenommen, Sie haben einen Screen mit 16 Farben, also 4 Planes. Nun wollen Sie Grafiken kopieren, die ausschließlich die Farbe mit der Registernummer 2 enthalten. Für einen solchen Grafikpunkt braucht lediglich des entsprechende Bit in der 1. Bitplane gesetzt zu sein (Zählung beginnt bei 0), da 2^1 gleich 2 ist. Hier reicht es also völlig aus, nur die 1. Bitplane zu kopieren, vorausgesetzt, die übrigen Planes im Grafik-Zielbereich enthalten nur Nullen.

Im Masken-Register d7 muß nun für jede Bitplane, die kopiert werden soll, das ihrer Nummer entsprechende Bit gesetzt werden. In obigem Beispiel müßte also das 1. Bit (für die 1. Plane) gesetzt werden, d.h. ins Register müßte eine 2 geschrieben werden. Im Falle der 1. und 3. Bitplane wäre es eine 10 usw.

Ins Register a2 muß ein Zeiger auf einen Zwischenspeicher eingetragen werden, der groß genug für eine Grafikzeile der Quellgrafik sein sollte. Er wird bei Überlagerungen der Quell- und Zielgrafik benötigt.

Nun zur dritten Blitterroutine:

BltBitMapRastPort	=	-606 (Graphics-Library)

*srcbm	a0	<	Zeiger auf Quell-Bitmap
srcx	d0	<	x-Startkoordinate der Quellgrafik
srcy	d1	<	y-Startkoordinate der Quellgrafik
*destrp	a0	<	Zeiger auf Ziel-Rastport
destx	d2	<	x-Koordinate der Zielgrafik
desty	d3	<	y-Koordinate der Zielgrafik
sizex	d4	<	x-Größe der zu kopierenden Grafik
sizey	d5	<	y-Größe der zu kopierenden Grafik
minterm	d6	<	Logische Verknüpfung für die Kopie

Erklärung Kopiert einen rechteckigen Grafik-Ausschnitt aus einer Bitmap in einen Rastport.

Diese Routine stellt eine Transfer-Kopierroutine dar, die Grafiken aus einer Bitmap in einen Rastport kopiert. Im Grunde kann man sich natürlich auch aus dem Ziel-Rastport den Zeiger auf die Bitmap besorgen und dann BltBitMap benutzen. Diese Routine kann man einsetzen, wenn es programmtechnisch günstig ist.

7.6.2 Schnelles Löschen von Grafiken

Die letzte Blitter-Routine, dient nicht dem Kopieren, sondern dem schnellen Löschen von Grafiken und Speicherbereichen:

BltClear	=	-300 (Graphics-Library)

*memory	a1	<	Zeiger auf Beginn des zu löschenden Speicherbereichs
size	d0	<	Größe des zu löschenden Speicherbereichs
flags	d1	<	Bestimmt die Interpretation von 'size' und das Warteverhalten der Routine

Erklärung Löscht rechteckige Grafiken oder Speicherbereiche.

Im 'flags'-Register sind nur die zwei untersten Bits von Bedeutung. Wenn das Bit 0 gesetzt ist, wird das aktivierte Programm während des Löschvorgangs angehalten, die Routine kehrt also erst nach Beendigung des Löschens zurück. Ansonsten wird das Programm sofort fortgesetzt, das Ende des Löschvorgangs wird nicht abgewartet.

Das Bit 1 legt fest, wie die 'size'-Angabe in d0 zu interpretieren ist. Bei gelöschtem Bit gibt d0 die Gesamtlänge eines zusammenhängenden Speicherbereichs in Byte an. Ist das Bit gesetzt, so wird das obere Wort des Langworts d0 als Hö-

hen- und das untere Wort als Breitenangabe eines rechteckig-organisierten Speicherbereichs (allerdings in Byte, nicht in Pixeln) angesehen. Die Gesamt-Bytezahl berechnet BltClear aus der Multiplikation des oberen mit dem unteren Wort.

Das folgende Programm demonstriert die Benutzung von ClipBlit und BltBitMap. Es zeichnet zwei verschiedenfarbig ausgefüllte Kreise in ein Fenster und kopiert sie unter Anwendung verschiedener Minterms in ein zweites Fenster. Sie finden es unter "PRG_7_10.S" auf der Diskette.

```
* Programm 7.10 (Auszug): Demonstration ClipBlit und BltBitMap

        ...

        bsr       verz            ; Verzögerung per Delay

* 1:1-Kopie des Windows 1 ins Window 2

        move.l    a3,a0           ; Quell-Rastport
        moveq     #0,d0           ; Quell-Startkoordinaten
        moveq     #0,d1
        move.l    $32(a5),a1      ; Ziel-Rastport
        moveq     #0,d2           ; Ziel-Koordinaten
        moveq     #0,d3
        move.l    #300,d4         ; Breite und Höhe
        move.l    #100,d5         ; der Grafik
        move.b    #192,d6         ; Minterm-Modus = 1:1-Kopie
        jsr       ClipBlit(a6)

        bsr       verz

* Invertierte Kopie von Window 2 in Window 1

        move.l    $32(a5),a0      ; Quell-Rastport
        moveq     #0,d0           ; Quell-Koordinaten
        moveq     #0,d1
        move.l    a3,a1           ; Ziel-Rastport
        moveq     #0,d2           ; Ziel-Koordinaten
        moveq     #0,d3
        move.l    #300,d4         ; Breite und Höhe
        move.l    #100,d5
        move.b    #48,d6          ; Minterm-Modus = Invertiere Kopie
        jsr       ClipBlit(a6)

        bsr       verz

* OR-Verknüpfung der ersten Planes der beiden Windows

        move.l    $32(a4),a0      ; Quelle = Window 1
        move.l    4(a0),a0        ; Vom Rastport zur Bitmap
        moveq     #5,d0           ; Start-x = 5
        moveq     #15,d1          ; Start-y = 15
```

```
        move.l  $32(a5),a1      ; Ziel = Window 2
        move.l  4(a1),a1        ; Zur Bitmap
        move.l  #315,d2         ; Ziel-Koordinaten
        move.l  #120,d3
        move.l  #290,d4         ; Breite
        move.l  #70,d5          ; Höhe
        move.b  #224,d6         ; Minterm-Modus: OR-Verknüpfung
        move.b  #1,d7           ; Nur Plane 1 betroffen
        lea     buff,a2         ; Zwischenspeicher
        jsr     BltBitMap(a6)

        bsr     verz

* EOR-Verknüpfung

        move.l  a3,a0           ; Quelle = Window 1
        moveq   #0,d0           ; Koordinaten
        moveq   #0,d1
        move.l  $32(a5),a1      ; Ziel = Window 2
        moveq   #0,d2           ; Koordinaten
        moveq   #0,d3
        move.l  #300,d4         ; Breite
        move.l  #100,d5         ; Höhe
        move.b  #96,d6          ; Minterm-Modus: EOR-Verknüpfung
        jsr     ClipBlit(a6)
```

Programm 7.10 (Auszug)

Beachten Sie den Unterschied zwischen ClipBlit und BltBitMap bezugnehmend auf die Wahl der Koordinaten: Wie schon erwähnt sind die Koordinaten von ClipBlit relativ zum jeweiligen Rastport zu sehen, während sie bei BltBitMap relativ zur Screen-Oberkante sind.

7.6.3 Scrollen von Bildausschnitten

Unter "Scrolling" versteht man die pixelweise Verschiebung eines Grafikbereiches. Man könnte dafür eine Kopierroutine verwenden und die Koordinaten der Quell- und Zielgrafik entsprechend wählen. Graphics stellt uns aber auch eine spezielle Routine zur Verfügung:

ScrollRaster	=	**-396 (Graphics-Library)**

***rp**	**a1**	<	Zeiger auf Rastport, in dem gescrollt werden soll
dx	**d0**	<	Anzahl der Pixel, um die die Grafik in x-Richtung verschoben werden soll
dy	**d1**	<	Anzahl der Pixel, um die die Grafik in y-Richtung verschoben werden soll
minx	**d2**	<	x-Koordinate der linken oberen Ecke des Grafikausschnitts

miny	**d3**	<	y-Koordinate der linken oberen Ecke des Grafikausschnitts
maxx	**d4**	<	x-Koordinate der rechten unteren Ecke des Grafikausschnitts
maxy	**d5**	<	y-Koordinate der rechten unteren Ecke des Grafikausschnitts

Erklärung Scrollt (verschiebt) einen rechteckigen Grafikausschnitt um eine wählbare Anzahl Pixel.

Beachten Sie, daß die Werte für 'dx' und 'dy' angeben, um wieviel die Grafik nach links bzw. oben gescrollt wird. Soll sie nach rechts bzw. unten gescrollt werden, sind für dx und dy negative Werte anzugeben.

7.6.4 Multitasking-gerechtes Warten auf Graphics-Ebene

Im DOS-Kapitel haben wir eine Methode kennengelernt, die eine Verzögerung des Programms ohne Behinderung des Multitasking-Betriebs erreicht. Auf Graphics-Ebene gibt es ähnliche Fälle. Hier ist es oft nötig, bestimmte Funktionsaufrufe mit dem Bildaufbau zu koordinieren. So ist es z.B. recht ungünstig, Änderungen an Bildschirmgrafiken vorzunehmen, wenn der Rasterstrahl, der das Monitorbild aufbaut, gerade diese Grafikbereiche darstellt. Die einfachste Methode, um dies zu verhindern, ist, auf den Rücklauf des Strahl vom rechten unteren zum linken oberen Rand zu warten, und genau das tut die Routine WaitTOF:

WaitTOF	=	**-270 (Graphics-Library)**

Erklärung Wait for Top of Frame - wartet auf den Rücklauf des Elektronen-Rasterstrahls.

WaitTOF versetzt unser Programm solange in den Wartezustand und verhindert die Vergeudung von Prozessorzeit, bis der Strahlrücklauf durchgeführt wurde.

Es gibt noch eine weitere Rasterstrahl-Warteroutine, deren Funktionsweise Sie erst nach dem nächsten Abschnitt richtig verstehen können. Sie sei hier aber schon einmal vorgestellt:

WaitBOVP	=	**-402 (Graphics-Library)**

***vp**	**a0**	<	Zeiger auf ViewPort, auf dessen Ende gewartet werden soll

Erklärung Wait for Bottom of ViewPort - wartet, bis der Rasterstrahl die letzte Zeile

des angegebenen Viewports gezeichnet hat.

Als Beispielprogramm für ScrollRaster und WaitTOF eignet sich ein kleines Scrollschriftprogramm sehr gut. Es öffnet ein 12 Pixel hohes Window am unteren Rand der Workbench, zeichnet zwei waagerechte Linien und läßt einen Text im Fenster scrollen.

```
* Programm 7.11 (Auszug): Demonstration ScrollRaster

        ...

        lea     scrtext,a3      ; Textzeichen-Zeiger

main1:  move.l  gfxbase,a6
        moveq   #0,d6           ; Schleife: Jeden Buchstaben zweimal
                                ; ausgeben

main3:  move.l  a5,a1           ; Rastport
        move.l  #637,d0         ; x-Start des Textes auf 637
        tst.b   d6              ; Zweiter Schleifendurchlauf?
        beq     main2           ; Wenn nein
        subq    #4,d0           ; x-Start auf 633
main2:  moveq   #8,d1           ; y-Start auf 8 (relativ zum Window)
        jsr     Move(a6)        ; Zeichencursor setzen

        move.l  a5,a1           ; Ein Textzeichen ausgeben
        move.l  a3,a0
        moveq   #1,d0
        jsr     Text(a6)

        move.l  a5,a1           ; Rastport
        moveq   #4,d0           ; Scrolle 4 Pixel nach links
        moveq   #0,d1           ; Kein Scrolling nach oben
        moveq   #0,d2           ; x-Start bei 0
        moveq   #2,d3           ; y-Start bei 2
        move.l  #639,d4         ; x-Ende bei 639
        moveq   #10,d5          ; y-Ende bei 10
        jsr     ScrollRaster(a6)      ; Scrollen

        jsr     WaitTOF(a6)     ; Warte auf Strahlrücklauf

        addq    #1,d6           ; Schleife Textausgabe
        cmp.b   #2,d6
        bne     main3

        add.l   #1,a3           ; Zeichenzeiger erhöhen
        tst.b   (a3)            ; Ende-Nullbyte erreicht?
        bne     main4           ; Wenn nein
        lea     scrtext,a3      ; Scrolltext von vorne
```

```
main4:  move.l   ExecBase,a6

        move.l   $56(a4),a0      ; Message abholen
        jsr      GetMsg(a6)
        tst.l    d0              ; Keine da?
        beq      main1           ; Wenn so

        ...

scrtext:     dc.b    "Mein erster Scrolltext !  ",0
             even
```

Programm 7.11 (Auszug)

Beim Scrolling wird der Text in einer Schleife durchlaufen, wobei bei jedem Durchlauf ein Zeichen ausgegeben wird.

Ausgabe und Scrollen erfolgt für jedes Zeichen zweimal: Zuerst wird das Zeichen ganz am linken Rand ausgegeben, dann wird die ganze Zeile vier Pixel nach links gescrollt, anschließend wird das Zeichen vier Pixel vom linken Rand entfernt ausgegeben und die ganze Zeile nochmal um vier Pixel gescrollt. Vor jedem Ausgabeaufruf wird per WaitTOF auf den Rücklauf des Rasterstrahls (50 Mal pro Sekunde) gewartet. Das Monitorbild wird also alle 1/50 Sekunde von neuem aufgebaut.

Die zweifache Ausgabe jedes Zeichens hat den Zweck, das Scrolling nicht zu schnell werden zu lassen. Man könnte natürlich jedes Zeichen auch nur einmal ausgeben und die Zeile dann um 8 Pixel scrollen, dann jedoch würde die Schrift kaum noch lesbar sein. Will man aber gezielt hohe Geschwindigkeiten erreichen, gibt man jedes Zeichen nur einmal aus.

Die Geschwindigkeit kann aber auch herabgesetzt werden: Anstatt jedes Zeichen ein- oder zweimal auszugeben, könnte man dies auch viermal oder sogar achtmal tun. Bei viermaliger Ausgabe müßte man dann jedesmal um zwei Pixel scrollen, bei achtmaliger sogar nur um einen (Ergebnis: Scrollschrift im Schneckentempo).

7.7 Einrichten eigener Rast- und ViewPorts

Bisher haben wir nur Rast- und Viewports benutzt, die quasi von Intuition "vorgefertigt" waren. Es gibt aber auch die Möglichkeit, diese Strukturen selbst anzulegen und so einen von Screens und Windows unabhängigen Ausgabebildschirm zu erzeugen. Fangen wir dabei mit dem einfacheren Thema an: den Rastports.

7.7.1 Initialisierung einer neuen Rastport-Struktur

Dazu ist kein großer Aufwand nötig. Sie brauchen lediglich 100 Bytes für die Struktur zu reservieren (am besten mit 'ds.b') und die InitRastPort-Routine aufzurufen:

InitRastPort	=	**-198 (Graphics-Library)**

***rastPort**	a1	<	Zeiger auf Speicherbereich für zu initialisierende Rastport-Struktur
Erklärung			Richtet eine neue Rastport-Struktur ein und füllt sie mit den Standard-Werten.

Das war schon alles. Auf den so eingerichteten Rastport können Sie alle Graphics-Routinen, die mit Rastports arbeiten, anwenden. Es entsteht allerdings ein Problem: Die Grafik-Befehle, die Sie auf den Rastport ausführen, laufen nur "im Hintergrund" ab. Sehen können Sie die Grafik nicht, da sie ja zu keinem Screen oder Window gehört, sondern sozusagen "frei im Speicher schwebt". Um sie auf dem Bildschirm sichtbar zu machen, müssen Sie die Viewports benutzen. Dazu werden wir uns aber erst ein wenig Hintergrundwissen aneignen.

7.7.2 Die View- und ViewPort-Struktur

Einen ViewPort kann man als Graphics-Äquivalent eines Intuition-Screens ansehen. Er hat eigene Farben, eine eigene Auflösung und seine Startposition und Ausdehnung können festgelegt werden. Aus den Viewports berechnet die Graphics-Library Steuerprogramme für die Grafik-Hardware, wodurch die Grafik letztendlich erst sichtbar wird.

Interessant ist in diesem Zusammenhang, wie die Überlagerung von Intuition-Screens funktioniert. Intuition arbeitet auch mit Viewports; für jeden Screen wird ein solcher eingerichtet. Überlagern sich nun mehrere Screens, werden ihre Viewports entsprechend angepaßt. Ihre Positionen werden gemäß denen der Screens verschoben, und ihre vertikalen Größen werden so eingestellt, daß sie den sichtbaren Teilen ihrer zugehörigen Screens entsprechen. Viewports selbst dürfen sich nämlich nicht überlagern, sie müssen alle hübsch untereinander stehen. Diese Viewport-Liste wird dann auf dem Bildschirm angezeigt, wodurch sich eine scheinbare Überlagerung ergibt. Jetzt ist auch klar, warum das Ziehen eines Screens manchmal etwas zäh vor sich geht: Bei jeder Änderung an der Lage der Screens zueinander müssen die ganze Viewport-Liste und die daraus resultierenden Hardware-Strukturen neu berechnet werden, was eine Weile dauern kann.

Alle Viewports sind untereinander verkettet. Der erste enthält einen Eintrag, der auf den zweiten zeigt, dieser einen, der auf den dritten zeigt usw. Beim letzten Viewport ist dieser Zeiger 0. An den "Kopf" der Viewport-Liste wird nun eine sog. "View-Struktur" gestellt, die das ganze Viewport-Chaos verwaltet. Einen View kann man als die Zusammenfassung aller momentan auf dem Bildschirm sichtbaren Screens (sprich Viewports) ansehen. Die Struktur sieht so aus:

Die View-Struktur

```
00    dc.l   *v_ViewPort       ; Zeiger auf den ersten ViewPort
04    dc.l   *v_LOFCprList     ; Zeiger auf Longframe-Copperlist
08    dc.l   *v_SHFCprList     ; Zeiger auf Shortframe-Copperlist
12    dc.w    v_DyOffset       ; y-Startkoordinate des View
14    dc.w    v_DxOffset       ; x-Startkoordinate des View
16    dc.w    v_Modes          ; Bildschirmauflösung etc.
18            v_SIZEOF
```

***v_ViewPort**
Dieser Zeiger leitet die Viewport-Verkettung ein.

***v_LOFCprList**
Hier steht ein Zeiger auf eine CprList-Struktur, welche dem Aufbau der sogenannten "Copperliste" des View dient. Die Liste, auf die hier gezeigt wird, findet im interlace- und nicht-interlace-Modus des Bildschirms Anwendung. Der Copper ist ein Koprozessor, der in Abhängigkeit von der Position des Rasterstrahls bestimmte Hardwareregister verändern kann. Er ist in erster Linie für die Ansteuerung der Grafik-Hardware zuständig, so laufen die Einstellung der Farben, der Auflösung und der auszugebenden Grafik über ihn. Eine Copperliste ist nichts anderes, als ein Programm für den Koprozessor.

***v_SHFCprList**
Befindet sich der Bildschirm im interlace-Modus, werden zwei Copperlisten benötigt. In diesem Fall steht hier der Zeiger auf die zweite Liste.

v_DyOffset, v_DxOffset
Diese Einträge bestimmen die Startkoordinaten des View auf dem Bildschirm.

v_Modes
Die hier anzugebenden Auflösungs- und Bildschirmmodi entsprechen denen beim Öffnen eines Intuition-Screens. Die Viewmodes, die Sie hier angeben, gelten als erlaubte Modis für alle Viewports. Hier die Liste der Zahlenwerte, eine genaue Beschreibung finden Sie im Intuition-Kapitel.

View-Mode	Wert	Bedeutung
GENLOCK_VIDEO	$0002	Bindet eine externe Signalquelle ein
EXTRA_HALFBRITE	$0080	64-Farben Modus
DUALPF	$0400	Dual-Playfield
Hold-And-Modify	$0800	HAM-Modus (4096 Farben)
VP_HIDE	$2000	Kein Bild
SPRITES	$4000	View mit Hardware-Sprites
HIRES	$0004	Verdoppelt Auflösung in x-Richtung
LACE	$8000	Verdoppelt Auflösung in y-Richtung

Noch ein paar Worte zu den Startkoordinaten. Sie stimmen nicht mit den von Intuition gewohnten Koordinaten überein. 0/0 liegt also nicht da, wo man sonst die linke obere Ecke eines neuen Screens mit den Koordinaten 0/0 erwarten würde, sondern weit außerhalb des sichtbaren Bildschirmbereichs. Die von Intuition benutzten View-Startkoordinaten liegen etwa bei 40 in y- und 120 in x-Richtung. Diese Werte können, je nach Preferences-Bildeinstellung, etwas variieren. Der sichtbare Bereich beginnt bei 29 in y- und 93 in x-Richtung.

Nun wissen Sie auch, wie das Preferences-Programm in bezug auf die Bildzentrierung arbeitet. Es verändert einfach die Startkoordinaten des Intuition-View (in gewissen Grenzen). Außerdem fällt bei Betrachtung der von Intuition verwendeten Startkoordinaten und der Koordinaten, bei denen der sichtbare Bildbereich beginnt, auf, daß das Bild noch ein ganzes Stück links bzw. oberhalb der Intuition-Begrenzung sichtbar ist. Man könnte also das Bild nach links und oben verschieben, und es dafür ein Stück breiter und höher machen. Tatsächlich lassen sich so Grafiken darstellen, die über die Intuition-Begrenzung von 640 Punkten horizontal und 256 Punkten vertikal (im Modus Hires Non-Interlaced) hinausgehen. Dies nennt man Overscan-Modus.

Beim Anlegen eines eigenen View brauchen Sie sich im Normalfall nicht mit dem Ausprobieren der günstigsten Startkoordinaten herumzuschlagen, sie werden von der Initialisierungsroutine automatisch auf die Intuition-Standardwerte gesetzt (die Sie natürlich nachträglich ändern können).

Es gibt eine Graphics-Routine, die eine neue View-Struktur anlegt, d.h. diese mit Standard-Werten füllt. Sie brauchen später nur noch den Zeiger auf den ersten Viewport und den Viewmode einzutragen. Die Routine heißt InitView:

InitView	=	**-360 (Graphics-Library)**

***view** a1 < Zeiger auf Speicherbereich für neue View-Struktur

Erklärung Legt eine neue View-Struktur an und füllt sie mit Standardwerten.

Sie brauchen also lediglich 18 Bytes Speicher zu reservieren, InitView aufzurufen und später ViewPort und Modes zu setzen. Um die restlichen Einträge müssen Sie sich nicht kümmern (zumindest nicht direkt).

Nun kommen wir zu den eigentlichen Viewports. Ihre Struktur sieht folgendermaßen aus:

Die ViewPort-Struktur

```
00    dc.l   *vp_Next           ; Zeiger auf den nächsten ViewPort
04    dc.l   *vp_ColorMap       ; Zeiger auf ColorMap-Struktur
08    dc.l   *vp_DspIns         ; Zeiger auf Display-Copperliste
12    dc.l   *vp_SprIns         ; Zeiger auf Sprite-Copperliste
16    dc.l   *vp_ClrIns         ; Zeiger auf Color-Copperliste
20    dc.l   *vp_UCopIns        ; Zeiger auf User-Copperliste
24    dc.w    vp_DWidth         ; Breite des ViewPorts
26    dc.w    vp_DHeight        ; Höhe des ViewPorts
28    dc.w    vp_DxOffset       ; x-Startkoordinate
30    dc.w    vp_DyOffset       ; y-Startkoordinate
32    dc.w    vp_Modes          ; Bildschirmauflösung etc.
34    dc.w    vp_reserved       ; Reserviert für Erweiterungen
36    dc.l   *vp_RasInfo        ; Zeiger auf RasInfo-Struktur
40            vp_SIZEOF
```

***vp_Next**
Wie gesagt sind alle Viewports eines View untereinander verkettet. An dieser Stelle steht jeweils ein Zeiger auf den nächsten Viewport, im letzten steht hier eine 0.

***vp_ColorMap**
Die ColorMap-Struktur gibt die Farben für den Viewport an. Zu ihr kommen wir gleich.

***vp_DspIns - *vp_UCopIns**
Diese Zeiger weisen auf CopList-Strukturen, welche dem Aufbau der Teil-Copperlisten des Viewport dienen. Es gibt getrennte Copperlisten für die Bildschirmauflösung, die Sprites (falls vorhanden) und die Farben. In vp_UCopIns können Sie eine eigene Copperliste einbinden (User-Copperlist). Über eine Graphics-Funktion werden die Teillisten aller Viewports zur Gesamtliste des übergeordneten View zusammengefaßt.

vp_DWidth, vp_DHeight
Die Höhe und Breite des Viewports. Diese Werte können auch kleiner sein als die voll auszugebende Grafik, dann fehlen einfach die entsprechenden Teile an den Seiten.

vp_DxOffset, vp_DyOffset
Diese Einträge geben die horizontale und vertikale Startkoordinate des Viewports an. Sie sind relativ zu den Offsets des übergeordneten View zu sehen und können auch negativ sein (dann beginnt der Viewport links bzw. oberhalb vom Haupt-View).

vp_Modes
Die View-Modi des Viewports. Erlaubt sind nur die Modi, die im Modes-Eintrag des View angegeben sind.

***vp_RasInfo**
Zeiger auf eine Struktur, welche die Verbindung zwischen Viewport und den eigentlichen Grafikdaten im Speicher herstellt.

Auch zur Initialisierung dieser Struktur gibt es eine Graphics-Routine:

InitVPort	=	-204 (Graphics-Library)

***viewPort**	**a0**	<	Zeiger auf Speicherplatz für die neue ViewPort-Struktur
Erklärung			Richtet eine neue ViewPort-Struktur ein und belegt sie mit den Standard-Werten.

Die durch diese Routine eingestellten Standardwerte können Sie natürlich auch nachträglich ändern, z.B. die Positions-Offsets oder die Viewmodi. Selbst belegen müssen Sie die Einträge vp_DWidth, vp_DHeight, vp_ColorMap und vp_RasInfo, da dies nicht von InitVPort übernommen wird.

ColorMap, BitMap und RasInfo

Kommen wir nun zur nächsten benötigten Struktur, der ColorMap-Struktur. Ihre Einrichtung brauchen Sie wie gewohnt nicht von Hand zu übernehmen, trotzdem sei sie hier vorgestellt:

Die ColorMap-Struktur

```
00    dc.b    cm_Flags          ; Interne Flags
01    dc.b    cm_Type           ; Betriebssystemsversion
02    dc.w    cm_Count          ; Anzahl der Farbeinträge
04    dc.l   *cm_ColorTable     ; Zeiger auf Farbtabelle
08            cm_SIZEOF
```

cm_Flags
Interne Flags, für den Programmierer unwichtig (im Zweifelsfall auf 0 setzen).

cm_Type
Bei der Betriebssystemsversion 1.2 oder früher steht hier eine 0, bei späteren eine 1.

cm_Count
Anzahl der Farbeintrags-Worte in der Farbtabelle

cm_ColorTable
Ein Zeiger auf eine Wort-Tabelle, in der jedes Wort für einen Farbeintrag steht. Die Kodierung erfolgt wie im Abschnitt über die Farbpalette (LoadRGB4) besprochen.

Für die Einrichtung dieser Struktur ist die Routine GetColorMap zuständig:

GetColorMap	=	**-570 (Graphics-Library)**

entries	d0	<	Anzahl der gewünschten Farb-Einträge
***cm**	d0	>	Zeiger auf initialisierte ColorMap-Struktur oder 0, wenn der Speicherplatz nicht ausreichte
Erklärung			Reserviert Speicher für eine ColorMap-Struktur einschließlich Farbtabelle und richtet die Struktur ein.

Den Zeiger, den wir von dieser Routine erhalten, brauchen wir nur in den Eintrag vp_ColorMap (Offset 4) des Viewports einzutragen, und schon ist unser Viewport bereit für Farbänderungen. Diese können mit den schon bekannten Routinen SetRGB4, SetRGB4CM oder LoadRGB4 vorgenommen werden, genau wie bei Intuition-Screen-Viewports.

Nun wollen wir aber nicht nur Farben einstellen und Bildschirmparameter bestimmen. Wir wollen auch Grafik zeichnen, und für die brauchen wir einen Speicherbereich. Dieser Speicherbereich wird später von der Videohardware auf dem Bildschirm abgebildet, weshalb er im Chip-RAM liegen muß. Im Intuition-Kapitel haben wir schon etwas über den Bitplane-artigen Aufbau einer Amiga-Grafik erfahren. Zur Erinnerung: Eine Grafik besteht aus bis zu sechs Bitplanes, die alle die gleiche Spalten- und Zeilenanzahl haben. Die Farbtabellennummer eines Punktes ergibt sich als Kombinations-Binärzahl aus den "übereinanderliegenden" Punkten (Bits) aller Bitplanes.

Auf Graphics-Ebene ist das natürlich genauso. Für jede Bitplane, über die unsere Grafik verfügen soll, müssen wir gesondert Speicher reservieren und die Startadressen in einer neuen Struktur, der BitMap-Struktur, ablegen (die auch noch ein paar weitere Daten enthält):

Die BitMap-Struktur

```
00   dc.w   bm_BytesPerRow      ; Anzahl Bytes in einer Grafikzeile
02   dc.w   bm_Rows             ; Anzahl Zeilen in der Grafik
04   dc.b   bm_Flags            ; System-Flags
05   dc.b   bm_Depth            ; Tiefe, Anzahl Bitplanes
06   dc.b   bm_Pad              ; Reserviert für Erweiterungen
```

```
08      ds.l    *bm_Planes,8            ; 8 Langwort-Planezeiger
40               bm_SIZEOF
```

bm_BytesPerRow
Die Anzahl Bytes pro Zeile ergibt sich aus der Division der Pixelzahl pro Zeile durch 8 (jedes Bit repräsentiert ein Pixel, ein Byte hat 8 Bit).

bm_Rows
Die Anzahl Bytes pro Zeile muß mit der Anzahl Zeilen malgenommen werden, dann ergibt sich die Speichergröße, die ein Plane der Grafik benötigt

bm_Depth
Nimmt man diese Größe noch mit der Anzahl der Planes mal, hat man den Gesamtspeicherbedarf des Grafikschirms.

***bm_Planes**
Hier müssen die Zeiger auf die Bitplane-Startadressen (für jede Plane getrennt) eingetragen werden. Obwohl hier 8 Langworte vorgesehen sind, liegt die Maximalzahl Bitplanes in einer Grafik bei 6. (Vielleicht will Commodore in Zukunft einen 8-Bitplane-Amiga herausbringen ???)

Auch für diese Struktur gibt es eine Initialisierungs-Routine in der Graphics-Library:

InitBitMap	**=**	**-390**

***bitMap**	**a0**	<	Zeiger auf Speicherplatz für die zu initialisierende Bitmap-Struktur
depth	**d0**	<	Anzahl der Planes für die Bitmap
width	**d1**	<	Breite der Bitmap in Pixeln
height	**d2**	<	Höhe der Bitmap in Pixeln
Erklärung			Initialisiert eine Bitmap-Struktur mit den angegebenen Werten.

Den Speicher für die Planes reserviert man am besten mit AllocRaster (beschrieben im Abschnitt über die temporären Rastports, 7.3.5), da diese Routine die Umrechenarbeit Pixelhöhe/-breite in Bytes übernimmt und automatisch Chip-RAM anfordert.

Der Zeiger auf die somit erstellte Bitmap-Struktur muß in eine weitere Struktur, RasInfo, eingetragen werden, die so aussieht:

Die RasInfo-Struktur

```
00      dc.l    *ri_Next                ; Zeiger auf nächste RasInfo
04      dc.l    *ri_BitMap              ; Zeiger auf Bitmap
08      dc.w     ri_RxOffset            ; x-Koordinate des Rasters
```

```
10      dc.w    ri_RyOffset          ; y-Koordinate des Rasters
12              ri_SIZEOF
```

***ri_Next**
Wenn man mit Spezial-Grafikmodi wie Dual-Playfield arbeitet, werden mehrere Raster-Bitmaps für einen Viewport benötigt. Im Normalfall aber kann hier einfach eine 0 eingetragen werden.

***ri_BitMap**
Ein Zeiger auf eine Bitmap-Struktur. RasInfo dient somit als "Verbindungsstück" zwischen Viewport und Bitmap.

ri_RxOffset, ri_RyOffset
Auch dem Raster-Bitmap kann eine Startkoordinate gegeben werden. Das ist nützlich, wenn Sie nicht die komplette Grafik anzeigen wollen, sondern erst ab einer bestimmten Zeile und/oder Spalte. Im Normalfall können Sie einfach 0/0 eintragen.

In die RasInfo-Struktur wird ein Zeiger auf die Bitmap eingetragen, und in die Viewport-Struktur der Zeiger auf die RasInfo-Struktur (Eintrag vp_RasInfo, Offset 24). Die Verbindung zwischen Viewport und Bitmap ist somit hergestellt. Die RasInfo-Struktur ist die einzige Struktur in diesem Zusammenhang, für die es keine Initialisierungs-Routine in der Graphics-Library gibt. Sie müssen die Werte also "von Hand" mit MOVE-Befehlen in die Struktur schreiben.

Der Zeiger auf die Bitmap-Struktur muß schließlich auch noch in die Rastport-Struktur, in den Eintrag rp_Bitmap (Offset 4), geschrieben werden.

7.7.3 Vorbereitung des Views zur Bilddarstellung

Wenn Sie alle gewünschten und benötigten Einstellungen in View und Viewport vorgenommen haben, insbesondere die Einstellung von vp_DWidth und vp_DHeight gemäß der Größe der Grafik, die Sie anzeigen möchten, trennen Sie nur noch drei Routinen-Aufrufe vom eigenen Grafikschirm. Der erste ist MakeVPort:

MakeVPort	=	**-216 (Graphics-Library)**

***view**	a0	<	Zeiger auf View-Struktur, zu der der zu initialisierende Viewport gehört
***viewPort**	a1	<	Zeiger auf zu initialisierenden Viewport
Erklärung			Berechnet die Teil-Copperlisten eines Viewports und stellt sie in CopList-Strukturen zusammen.

Wie diese Routine genau arbeitet, insbesondere wie die CopList-Struktur aussieht, ist hier nicht von Interesse. Sie müssen nur wissen, daß diese Routine aufzurufen ist, um aus den Angaben für die Startkoordinaten und die Größe eines Viewports sowie die Lage seiner Bitmap im Speicher, die Teil-Copperlisten für den Viewport zu berechnen. Die nächste Routine faßt alle Teil-Copperlisten der Viewports in einer Gesamt-Copperliste für den View zusammen:

MrgCop	=	**-210 (Graphics-Library)**

***view**	a1	<	Zeiger auf den View, dessen Gesamt-Copperliste berechnet werden soll
Erklärung			Faßt alle Teil-Copperlisten der Viewports zur View-Gesamt-Copperliste zusammen und bringt sie in eine CprList-Struktur.

Vielleicht interessiert Sie der Unterschied zwischen einer CopList- und einer CprList-Struktur. In der CopList-Struktur stehen die Copper-Befehle in einer Graphics-spezifischen Form mit diversen System-Informationen wie Zeigern auf den nächsten Befehl etc. In der CprList aber steht nur das reine Copper-Programm, das der Coprozessor direkt ausführen kann. Nun zur dritten und letzten Routine:

LoadView	=	**-222 (Graphics-Library)**

view	a1	<	Darzustellender View
Erklärung			Stellt den angegebenen View auf dem Bildschirm dar.

Diese Routine sorgt also dafür, daß unser vorbereiteter und Copper-berechneter View auf dem Bildschirm ausgegeben wird.

Falls Sie den Sinn der letzten drei Routinen noch nicht so ganz begriffen haben, macht nichts, rufen Sie sie einfach nach Beendigung Ihrer Viewport-Vorbereitungen oder -Änderungen nacheinander auf, dann kann nichts schiefgehen.

Damit nach Beendigung des Programms wieder die Grafik erscheint, die vorher zu sehen war, sollte man sich am besten irgendwann vor dem LoadView-Aufruf die Adresse des aktuellen Views speichern. Dies geschieht mit der Routine ViewAddress:

ViewAddress	=	**-294 (Intuition-Library)**

***view** d0 > Zeiger auf den aktuellen View

Erklärung Ermittelt einen Zeiger auf die Struktur des derzeit angezeigten View.

Beachten Sie, daß sich diese Routine in der Intuition-Library befindet! Vor dem Programmende können Sie dann, um die alte Grafik wieder erscheinen zu lassen, LoadView mit der von ViewAddress gemeldeten Adresse aufrufen.

7.7.4 "Aufräumarbeiten" vor dem Programmende

Im Laufe der View- und Viewport-Vorbereitungen haben wir eine ganze Menge Routinen aufgerufen, die einige Strukturen angelegt haben. Als "ordentliche" Programmierer sollten wir diese Strukturen vor dem Programmende natürlich auch wieder "aufräumen", sprich den belegten Speicherplatz freigeben. Das dürfen Sie aber erst tun, wenn der View durch Aufruf von LoadView mit der alten View-Adresse vom Bildschirm entfernt worden ist!

Beginnen wir mit dem Speicherbereich des Grafikrasters, den wir mit AllocRaster definiert und in die Bitmap-Struktur eingetragen haben. Mit der Routine FreeRaster, die bereits im Abschnit temporäre Rastports besprochen wurde, können wir den Raster wieder freigeben.

Als nächstes nehmen wir uns die, von GetColorMap angelegte Colormap-Struktur vor. Sie ist recht einfach zu entfernen:

FreeColorMap	=	**-576 (Graphics-Library)**

***colorMap** a0 < Zeiger auf freizugebende Colormap

Erklärung Gibt den, von der angegebenen Colormap-Struktur belegten Speicherbereich frei.

Anschließend behandeln wir die Teil- und Gesamt-Copperlisten. In den Viewports wurden durch MakeVPort vier CopList-Strukturen angelegt. Diese müssen durch einen Aufruf der folgenden Routine freigegeben werden:

FreeVPortCopLists	=	**-540 (Graphics-Library)**

***viewPort** a0 < Zeiger auf Viewport, dessen Teil-Copperlisten freigegeben werden sollen

Erklärung Gibt den, von den vier Teil-Copperlisten des angegebenen Viewports belegten Speicher frei.

Diese Routine muß für alle Viewports, die Sie benutzt haben, aufgerufen werden. In diesem Zusammenhang sei noch eine andere Routine, die auch zum Freigeben von CopLists dient, vorgestellt:

FreeCopList	=	**-546 (Graphics-Library)**

***copList**	**a0**	<	Zeiger auf freizugebende CopList

Erklärung Gibt den von einer CopList belegten Speicher frei.

Anstatt FreeVPortCopLists, die sich automatisch alle vier CopLists eines Viewport vornimmt, können Sie FreeCopList benutzen, was allerdings etwas mehr an Arbeit bedeutet. Dieser Routine müssen Sie nämlich die Inhalte der vier Copperlist-Einträge vp_DspIns bis vp_UCopIns Ihrer Viewports übergeben. FreeCopList ist nur wirklich sinnvoll, wenn Sie eigene CopList-Strukturen aufgebaut hätten.

Jetzt kommen die beiden Gesamt-Copperlisten aus der View-Struktur an die Reihe. Zeiger auf sie sind zu finden in den Einträgen v_LOFCprList (Offset 4) und v_SHFCprList (Offset 8). Die Inhalte dieser Einträge müssen der folgenden Routine übergeben werden:

FreeCprList	=	**-564 (Graphics-Library)**

***cprlist**	**a0**	<	Zeiger auf freizugebende CprList

Erklärung Gibt den von einer CprList-Struktur belegten Speicherplatz frei.

Beachten Sie den Unterschied zwischen CopList und CprList! Im Viewport stehen CopLists, im View aber CprLists, die mit unterschiedlichen Routinen zu entfernen sind.

7.7.5 Vorgehensweise bei der Arbeit mit Views

Da es wohl etwas schwierig ist, all die Informationen der letzten Abschnitte zu behalten, hier noch einmal eine Kurz-Zusammenstellung der für eine View-Erstellung notwendigen Schritte:

1. Mit 'ds.b' (leeren) Speicherbereich für die Strukturen RastPort (100 Bytes), View (18 Bytes), ViewPort (40 Bytes), RasInfo (12 Bytes) und BitMap (40 Bytes) im Programm reservieren.
2. RastPort, ViewPort und BitMap durch Aufruf der Routinen InitRastPort, InitVPort und InitBitMap (letztere mit den Angaben für Breite, Höhe und Tiefe der Grafik) initialisieren.
3. Mit AllocRaster Speicher für jede Bitplane der Grafik reservieren und die Zeiger auf sie in die BitMap-Struktur (Eintrag bm_Planes) eintragen.
4. Von GetColorMap eine ColorMap-Struktur einrichten lassen, den Zeiger auf die Struktur in den Eintrag vp_ColorMap des Viewport schreiben und die Farben mit SetRGB4, SetRGB4CM oder LoadRGB4 einstellen.
5. Die RasInfo-Struktur durch Belegen der Einträge ri_BitMap, ri_DxOffset und ri_DyOffset einrichten und den Zeiger auf sie in den Viewport eintragen (vp_RasInfo).
6. Den Eintrag rp_BitMap des Rastports mit dem Zeiger auf die Bitmap belegen.
7. Die View-Struktur mit InitView initialisieren und den Zeiger auf den (ersten) Viewport darin eintragen.
8. Bei Verwendung mehrerer Viewports die Schritte 1-3 wiederholen und die Viewports durch Eintragen der Zeiger auf den jeweils nächsten untereinander verketten.
9. Die Einträge vp_DWidth, vp_DHeight, vp_Modes und v_Modes mit den gewünschten Werten belegen.
10. Wenn gewünscht, die Einträge v_DxOffset, v_DyOffset, vp_DxOffset und vp_DyOffset nach Bedarf ändern.
11. Die Routine MakeVPort für alle Viewports aufrufen.
12. Die Routine MrgCop aufrufen.
13. Die Adresse des alten Views mit ViewAddress holen und merken.
14. Mit LoadView den neuen View auf den Bildschirm bringen.

Zur ordentlichen Entfernung eines View sind folgende Schritte notwendig:

1. Zurückholen des alten, gemerkten View per LoadView mit der alten View-Adresse.
2. Freigabe der Teil-Copperlisten aller Viewports durch Aufruf von FreeVPortCopLists.
3. Freigabe der View-Gesamtcopperlisten mit FreeCprList.
4. Freigabe des Colormap-Speichers mit FreeColorMap.
5. Freigabe des Grafik-Rasters mit FreeRaster.

Übrigens: Vielleicht wundert es Sie, daß zwischen Rastport und Viewport überhaupt keine direkte Zeiger-Verbindung besteht. Dies ist aber auch nicht nötig, da beide mit der Bitmap, die ja die eigentlichen Grafikdaten repräsentiert, in Verbindung stehen. Über den Rastport wird in die Bitmap gezeichnet, und über den Viewport wird selbige auf dem Bildschirm dargestellt. Alles klar?

Nun haben wir endlich alles beisammen, um einen eigenen, Intuition-unabhängigen Grafikbildschirm erstellen, anzeigen und wieder entfernen zu können. Das war sicherlich ein hartes Stück Theorie, aber das Thema Views und Viewports ist halt nicht so einfach.

7.7.6 Beispielprogramme

Die Mühe hat sich aber bestimmt gelohnt, denn die nun folgenden Beispielprogramme sind sehr interessant und teilweise auch recht witzig. Fangen wir gleich zünftig an mit einem "Workbench-Erdbeben-Programm". Es verändert zufallsgesteuert die Bild-Startkoordinate des Intuition-View in bestimmten Grenzen (maximal 2 Pixel über oder unter der richtigen Einstellung) und bewirkt so einen recht "erschütternden" Effekt:

```
* Programm 7.12: "Bench-Quake" (Erdbeben auf der Workbench)

ExecBase      =       4
OpenLib       =       -552
CloseLib      =       -414
ViewAddress   =       -294
RemakeDisplay =       -384
WaitTOF       =       -270

        move.l  ExecBase,a6     ; Libs öffnen
        lea     intname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,intbase
        lea     gfxname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,gfxbase

        move.l  intbase,a6

        jsr     ViewAddress(a6)        ; Adresse des Intui-View holen
        move.l  d0,a3           ; und sichern

        move.w  12(a3),d3       ; y-Koordinate des View
        move.w  14(a3),d4       ; x-Koordinate

main1:  move.w  d3,d1           ; Alte y-Koordinate nach d1
        bsr     random          ; Zufallszahl erzeugen (-2 bis +2)
        add.w   d0,d1           ; Zur y-Koordinate hinzuaddieren
        move.w  d4,d2           ; Selbiges für die x-Koordinate
        bsr     random
        add.w   d0,d2
        move.w  d1,12(a3)       ; Zufällig geänderte Koordinaten in den
        move.w  d2,14(a3)       ; View eintragen
```

```
        jsr     RemakeDisplay(a6)     ; Display neu berechnen lassen

        move.l  gfxbase,a6
        jsr     WaitTOF(a6)     ; Auf Strahlrücklauf warten

        move.l  intbase,a6

        btst    #6,$bfe001      ; Linke Maustaste gedrückt?
        bne     main1           ; Wenn nein
        btst    #2,$dff016      ; Rechte Taste gedrückt?
        bne     main1           ; Wenn nein

        move.w  d3,12(a3)       ; Alte Viewkoordinaten wiederherstellen
        move.w  d4,14(a3)       ; und das Display neu berechnen lassen
        jsr     RemakeDisplay(a6)

        move.l  ExecBase,a6

        move.l  intbase,a1      ; Libs schließen und Ende
        jsr     CloseLib(a6)
        move.l  gfxbase,a1
        jsr     CloseLib(a6)

        rts

random: move.l  a5,d0           ; Zufallszahlen-Generator:
        asl.l   #1,d0           ; Erzeugt eine Zahl zwischen -2 und +2
        bhi     rn1             ; und gibt sie in d0 zurück
        eor.l   #$1d872b41,d0
rn1:    move.l  d0,a5
        and.l   #$ffff,d0
        divu    #5,d0
        swap    d0
        and.l   #$ffff,d0
        not.w   d0
        add.w   #2,d0
        rts

* Datenbereich

intname:        dc.b    "intuition.library",0
                even
gfxname:        dc.b    "graphics.library",0
                even
intbase:        dc.l    0
gfxbase:        dc.l    0
```

Programm 7.12: "Bench-Quake" (Erdbeben auf der Workbench)

Die ursprünglichen Einstellungen für die Startkoordinaten werden aus der Intui-View-Struktur ausgelesen und in d3 und d4 zwischengespeichert. Bei jedem Durchlauf der Hauptschleife wird zu diesen Koordinaten eine Zahl zwischen +2 und -2 hinzuaddiert, und die neuen Koordinaten werden in die

View-Struktur eingetragen. Nach Aufruf von RemakeDisplay wird der View mit den neuen Einstellungen dargestellt. Abgebrochen werden kann die Schleife durch gleichzeitiges Drücken beider Maustasten. Vor dem Programmende werden dann die "richtigen" Startkoodinaten wieder eingetragen.

Das nächste Demoprogramm initialisiert einen kompletten, eigenen View. Auf diesem wird dann das schon bekannte Ellipsen-Muster dargestellt:

```
* Programm 7.13: Einrichtung eines Views

ExecBase       =       4
OpenLib        =       -552
CloseLib       =       -414
InitRastPort   =       -198
InitBitMap     =       -390
InitVPort      =       -204
InitView       =       -360
AllocRaster    =       -492
GetColorMap    =       -570
LoadRGB4       =       -192
MakeVPort      =       -216
MrgCop         =       -210
ViewAddress    =       -294
LoadView       =       -222
ClearScreen    =       -48
DrawEllipse    =       -180
FreeVPortCL    =       -540
FreeCprList    =       -564
FreeColorMap   =       -576
FreeRaster     =       -498

        move.l  4,a6            ; Libs öffnen
        lea     intname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,intbase
        lea     gfxname,a1
        clr.l   d0
        jsr     OpenLib(a6)
        move.l  d0,gfxbase

        move.l  gfxbase,a6

        lea     rport,a1        ; Rastport einrichten
        jsr     InitRastPort(a6)

        lea     vport,a0        ; Viewport einrichten
        jsr     InitVPort(a6)

        lea     bmap,a0         ; Bitmap mit den Werten
        moveq   #2,d0           ; Tiefe = 2
```

```
move.l   #320,d1          ; Breite = 320
move.l   #256,d2          ; Höhe = 256
jsr      InitBitMap(a6)   ; einrichten

lea      bmap,a4          ; Start der Bitmap-Struktur

move.l   #320,d0          ; Speicher für Plane 1
move.l   #256,d1
jsr      AllocRaster(a6)         ; allokieren
move.l   d0,8(a4)         ; Zeiger auf Speicher in Bitmap-
                          ; Struktur eintragen

move.l   #320,d0          ; Dasselbe mit Plane 2
move.l   #256,d1
jsr      AllocRaster(a6)
move.l   d0,12(a4)

moveq    #4,d0            ; Colormap für 4 Farben holen
jsr      GetColorMap(a6)
lea      vport,a0         ; Zeiger auf sie in Viewport
move.l   d0,4(a0)         ; eintragen

lea      coltab,a1        ; Zeiger auf Farbtabelle
moveq    #2,d0            ; Zwei Farben setzen
jsr      LoadRGB4(a6)

lea      bmap,a0          ; Bitmap-Zeiger
lea      rport,a1         ; Rastport-Zeiger
move.l   a0,4(a1)         ; Bitmap in Rastport eintragen
lea      rinfo,a1         ; Rasinfo-Zeiger
move.l   a0,4(a1)         ; Bitmap in Rasinfo eintragen
lea      vport,a0         ; Viewport-Zeiger
move.l   a1,$24(a0)       ; Rasinfo in Viewport eintragen

lea      view,a1          ; View-Struktur einrichten
jsr      InitView(a6)

lea      view,a0          ; View-Zeiger
lea      vport,a1         ; Viewport-Zeiger
move.l   a1,(a0)          ; Viewport in View eintragen

move.w   #320,$18(a1)     ; Breite des Viewport = 320
move.w   #256,$1a(a1)     ; Höhe des Viewport = 256
move.w   #0,$20(a1)       ; Viewmode des Viewport = 0
move.w   #0,$10(a0)       ; Viewmode des View = 0

jsr      MakeVPort(a6)    ; Teil-Copperlisten berechnen

lea      view,a1          ; View-Gesamtcopperlisten
jsr      MrgCop(a6)       ; berechnen

move.l   intbase,a6

jsr      ViewAddress(a6)         ; alte View-Adresse holen
move.l   d0,oldview       ; und merken
```

```
        move.l  gfxbase,a6

        lea     rport,a1        ; Rastport löschen
        jsr     ClearScreen(a6)

        lea     view,a1         ; Neuen View darstellen
        jsr     LoadView(a6)

        lea     rport,a4

main1:  move.l  a4,a1           ; Ellipsen-Muster zeichnen

        ...

finish: btst    #6,$bfe001      ; Linke Maustaste gedrückt?
        bne     finish          ; Wenn nein

        move.l  oldview,a1      ; Gemerkten View wieder darstellen
        jsr     LoadView(a6)

        lea     vport,a0        ; Teil-Copperlisten freigeben
        jsr     FreeVPortCL(a6)

        lea     view,a3         ; Gesamt-Copperlisten freigeben
        move.l  4(a3),a0
        jsr     FreeCprList(a6)
        move.l  8(a3),a0
        jsr     FreeCprList(a6)

        lea     vport,a0        ; Colormap freigeben
        move.l  4(a0),a0
        jsr     FreeColorMap(a6)

        lea     bmap,a3         ; Zeiger auf Bitmap
        move.l  8(a3),a0        ; Speicher für Plane 1 freigeben
        move.l  #320,d0
        move.l  #256,d1
        jsr     FreeRaster(a6)
        move.l  12(a3),a0       ; Speicher für Plane 2 freigeben
        move.l  #320,d0
        move.l  #256,d1
        jsr     FreeRaster(a6)

        move.l  ExecBase,a6

        move.l  gfxbase,a1      ; Libs schließen und Tschüß
        jsr     CloseLib(a6)
        move.l  intbase,a1
        jsr     CloseLib(a6)
        rts

* Datenbereich

intname:        dc.b    "intuition.library",0
                even
gfxname:        dc.b    "graphics.library",0
```

```
            even
intbase:    dc.l    0
gfxbase:    dc.l    0

rport:      dcb.b   100,0
view:       dcb.b   18,0
vport:      dcb.b   40,0
rinfo:      dcb.b   12,0
bmap:       dcb.b   40,0

oldview:    dc.l    0

coltab:     dc.w    $000,$ff0

xrad:       dc.l    140
yrad:       dc.l    0
```

Programm 7.13: Einrichtung eines Views

Die Funktionsweise dieses Programms dürfte aufgrund der vorangegangenen ausführlichen Beschreibung klar sein.

Kommen wir zum nächsten Programm. Es richtet keinen kompletten View, sondern nur einen Viewport ein und hängt diesen in die Liste der Intuition-Viewports, sprich der Screens. Dadurch erscheint der neue Grafikbereich vor den Intuition-Screens, ist selbst aber kein solcher. Auf diesem Viewport lassen wir eine Scrollschrift laufen, wie schon aus einem der vorangegangenen Programm bekannt.

```
* Programm 7.14: Scrollschrift auf eigenem Viewport

ExecBase      =     4
OpenLib       =     -552
CloseLib      =     -414
InitRastPort  =     -198
InitBitMap    =     -390
InitVPort     =     -204
AllocRaster   =     -492
GetColorMap   =     -570
LoadRGB4      =     -192
MakeVPort     =     -216
MrgCop        =     -210
ViewAddress   =     -294
LoadView      =     -222
ClearScreen   =     -48
Move          =     -240
Text          =     -60
ScrollRaster  =     -396
WaitTOF       =     -270
RemakeDisplay =     -384
FreeVPortCL   =     -540
```

```
FreeCprList   =        -564
FreeColorMap  =        -576
FreeRaster    =        -498

        move.l   4,a6            ; Libs öffnen
        lea      intname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,intbase

        lea      gfxname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,gfxbase

        move.l   gfxbase,a6

        lea      rport,a1        ; Neuen Rastport einrichten
        jsr      InitRastPort(a6)

        lea      vport,a0        ; Neuen Viewport einrichten
        jsr      InitVPort(a6)

        lea      bmap,a0         ; Bitmap-Struktur mit gewünschten
        moveq    #2,d0           ; Werten initialisieren
        move.l   #640,d1
        move.l   #14,d2
        jsr      InitBitMap(a6)

        lea      bmap,a4

        move.l   #640,d0         ; Speicher für Plane 1 allokieren
        move.l   #14,d1
        jsr      AllocRaster(a6)
        move.l   d0,8(a4)        ; und in Bitmap-Struktur eintragen

        move.l   #640,d0         ; Selbiges für Plane 2
        move.l   #14,d1
        jsr      AllocRaster(a6)
        move.l   d0,12(a4)

        moveq    #4,d0           ; Colormap-Struktur für 4 Farben
        jsr      GetColorMap(a6)       ; holen
        lea      vport,a0        ; und in Viewport eintragen
        move.l   d0,4(a0)

        lea      coltab,a1       ; Farben per LoadRGB4 setzen
        moveq    #2,d0
        jsr      LoadRGB4(a6)

        lea      bmap,a0         ; Bitmap nach a0
        lea      rport,a1        ; Rastport nach a1
        move.l   a0,4(a1)        ; Bitmap in Rastport eintragen
        lea      rinfo,a1        ; RasInfo nach a1
        move.l   a0,4(a1)        ; Bitmap in Rasinfo eintragen
        lea      vport,a0        ; Viewport nach a0
```

```
        move.l   a1,$24(a0)       ; Rasinfo in Viewport eintragen

        lea      vport,a1         ; Viewport holen
        move.w   #640,$18(a1)     ; Breite eintragen
        move.w   #14,$1a(a1)      ; Höhe eintragen
        move.w   #$8000,$20(a1)   ; Viewmode HIRES
        move.w   #245,$1e(a1)     ; y-Start-Koordinate

        move.l   intbase,a6

        jsr      ViewAddress(a6)        ; Intui-View holen
        move.l   d0,a5

        move.l   gfxbase,a6

        lea      vport,a1         ; Unser Viewport
        move.l   (a5),oldvport    ; Alten Intui-Viewport merken
        move.l   (a5),(a1)        ; Unseren Viewport in Intui-View
        move.l   a1,(a5)          ; einklinken

        jsr      MakeVPort(a6)    ; Teil-Copperlisten berechnen

        move.l   a5,a1            ; Gesamt-Copperlisten berechnen
        jsr      MrgCop(a6)

        lea      rport,a1         ; Rastport löschen
        jsr      ClearScreen(a6)

        move.l   a5,a1            ; Intui-View neu darstellen
        jsr      LoadView(a6)

        lea      scrtext,a3       ; Bekannte Scroll-Routine
        lea      rport,a4

main1:  move.l   gfxbase,a6

        ...

        move.l   oldvport,(a5)    ; Alten Intui-Viewport in View

        move.l   intbase,a6       ; Display neu berechnen
        jsr      RemakeDisplay(a6)

        lea      vport,a0         ; Viewport-Copperlisten freigeben
        jsr      FreeVPortCL(a6)

        lea      vport,a0         ; Colormap freigeben
        move.l   4(a0),a0
        jsr      FreeColorMap(a6)

        lea      bmap,a3          ; Bitmap-Speicher freigeben
        move.l   8(a3),a0
        move.l   #640,d0
        move.l   #14,d1
        jsr      FreeRaster(a6)
        move.l   12(a3),a0
```

```
        move.l  #640,d0
        move.l  #14,d1
        jsr     FreeRaster(a6)

        move.l  ExecBase,a6     ; Libs schließen und Ende

        move.l  gfxbase,a1
        jsr     CloseLib(a6)
        move.l  intbase,a1
        jsr     CloseLib(a6)
        rts

* Datenbereich

intname:        dc.b    "intuition.library",0
                even

gfxname:        dc.b    "graphics.library",0
                even
intbase:        dc.l    0
gfxbase:        dc.l    0

rport:          dcb.b   100,0
vport:          dcb.b   40,0
rinfo:          dcb.b   12,0
bmap:           dcb.b   40,0

coltab:         dc.w    $000,$ff0

scrtext:        dc.b    "Scrolltext auf eigenem Viewport !!!    ",0
                even

oldvport:       dc.l    0
```

Programm 7.14: Scrollschrift auf eigenem Viewport

7.8 Grafik-Elemente (Gels)

Wenn Sie sich schon einmal mit Actionspielen befaßt haben, dann haben Sie sicherlich reichlich Bekanntschaft mit den Gels (Graphic-Elements) gemacht. Es handelt sich dabei um (meist relativ kleine) schnelle, bewegliche Grafiken, z.B. Raumschiffe, Gegner oder Schüsse. Es gibt drei verschiedene Typen von GELS, von denen sich zwei sehr ähneln, die dritte aber völlig anderer Art ist. Befassen wir uns zunächst mit dem einfachsten GEL-Typ: den SimpleSprites.

7.8.1 Die SimpleSprites

Der Begriff "Sprite" ist im Computer-Jargon weit verbreitet. Die wörtliche Übersetzung lautet "Elfe" oder "Fee". Wie man

auf diesen Ausdruck für diese Art Grafik-Elemente gekommen ist, ist wohl etwas rätselhaft. Die Sprites werden nämlich gewöhnlich zur Darstellung von Schüssen oder Explosionen eingesetzt, was wohl nicht sehr "feenhaft" ist (außer wir haben es mit einer bösen Fee zu tun...).

Bei den Sprites handelt es sich um kleine Grafik-Elemente, die direkt von der Hardware erzeugt und in die normale Grafikausgabe eingemischt werden. Sie sind dementsprechend schnell zu bewegen und im Aussehen zu ändern. Ein solcher Geschwindigkeitsvorteil bringt aber gewöhnlich anderweitige Nachteile mit sich, so auch hier: Ein Sprite darf höchstens 16 Pixel breit, dafür aber beliebig hoch sein. Außerdem kann er höchstens vier verschiedene Farben annehmen (wobei eine Farbe transparent dargestellt wird), bzw. 16 Farben, wenn man zwei Sprites zusammenmischt.

Die Grafik-Hardware stellt uns 8 Sprites zur Verfügung (gezählt wird von 0 bis 7), von denen das Betriebssystem eins (das erste) für die Maus beschlagnahmt (richtig, die Maus ist ebenfalls ein Sprite). Über die übrigen 7 können wir, falls zur Zeit kein anderes sprite-benutzendes Programm läuft, frei verfügen.

Anmeldung eines SimpleSprite

Zunächst lassen wir uns den gewünschten Sprite vom System reservieren (falls er nicht schon reserviert ist), und zwar mit der Routine GetSprite:

GetSprite	=	**-408 (Graphics-Library)**

***sSprite**	**a0**	<	Zeiger auf SimpleSprite-Struktur für den zu benutzenden Sprite
spriteNum	**d0**	<	Nummer des gewünschten Sprites (0-7) oder -1, wenn der erste freie Sprite geholt werden soll
spriteNum	**d0**	>	Nummer des effektiv geholten Sprites oder -1, falls alle bzw. der gewünschte Sprite belegt waren
Erklärung			Reserviert einen bestimmten oder beliebigen Hardware-Sprite.

Die SimpleSprite-Struktur dient der Verwaltung des Sprites. Sie sieht folgendermaßen aus:

Die SimpleSprite-Struktur

```
00    dc.l   *ss_posctldata        ; Zeiger auf Grafik-Daten
04    dc.w    ss_height            ; Höhe
06    dc.w    ss_x                 ; x-Position
```

```
08      dc.w    ss_y                ; y-Position
10      dc.w    ss_num              ; Sprite-Nummer
12              ss_SIZEOF
```

Die Einträge, um die Sie sich selbst kümmern müssen, sind ss_posctldata und ss_height. Die übrigen werden automatisch vom System verwaltet, auch ss_num. In ss_height wird die Höhe des Sprites eingetragen und in ss_posctldata ein Zeiger auf die eigentlichen Grafik-Daten, zu deren Aufbau wir jetzt kommen wollen.

Da die Sprite-Grafikdaten von der Hardware auf den Bildschirm gebracht werden, müssen sie immer im Chip-RAM stehen. Die Daten sind wortweise aufgebaut. In die ersten beiden Worte trägt das System die Werte für die x- und y-Positionen ein, Sie können hier einfach zwei Nullen eintragen. Eine besondere Bedeutung hat lediglich das Bit 7 im zweiten Wort: Ist es gesetzt, handelt es sich um einen Attached-Sprite (mit 16 Farben). Dazu kommen wir später noch.

Dann folgen für jede Zeile des Sprites zwei Worte. Da Sprites, wie schon erwähnt, im Normalfalle vier Farben haben, werden zwei Bitplanes benötigt, um alle Farben darstellen zu können. Sprites sind immer 16 Punkte breit. Diese 16 Punkte haben genau in einem Wort Platz. Falls Sie einen schmaleren Sprite wollen, setzen Sie einfach die überflüssigen Punkte auf 0.

Die erste Farbe (Nummer 0) jedes Sprites gilt als die transparente Farbe. An den Stellen, die diese Farbe haben, wird die unter dem Sprite liegende Grafik sichtbar. Die zwei Worte für jede Zeile stellen die zwei Bitplanes dar. Die Farbnummer für jeden Punkt ergibt sich aus der Kombination der Plane-Bits, wobei das erste Wort in der Datenzeile immer für die untere Plane und das 2. für die obere steht. Folgende Farbkombinationen ergeben sich:

2. Wort	1. Wort	Farbe
0	0	durchsichtig
0	1	Farbe 1
1	0	Farbe 2
1	1	Farbe 3

Nach den Worten für die letzte Zeile müssen noch zwei Null-Worte als Endekennung folgen. Als Beispiel nun die Definition einer kleinen Pacman-Spritegrafik, die auch im nächsten Demoprogramm Verwendung finden wird:

```
ssprite:    dc.l    sprdata         ; Zeiger auf Daten
            dc.w    16              ; Höhe = 16 Pixel
            dc.w    0,0,0           ; Wird vom System verwaltet

            section "",data_c       ; Folgende Daten ins Chip-RAM
```

```
sprdata:    dc.w   0,0               ; Positionskontrolle (System)
            dc.w   %0001111111111000,%0001111111111000
            dc.w   %0011111111111100,%0010000000000100
            dc.w   %0111111100111110,%0100000011000010
            dc.w   %1111111100111111,%1000000011000011
            dc.w   %1111111111111100,%1000000000001100
            dc.w   %1111111111110000,%1000000000110000
            dc.w   %1111111111000000,%1000000011000000
            dc.w   %1111111100000000,%1000000100000000
            dc.w   %1111111111000000,%1000000011000000
            dc.w   %1111111111111000,%1000000000111000
            dc.w   %1111111111111111,%1000000000000111
            dc.w   %1111111111111111,%1000000000000001
            dc.w   %0111111111111110,%0100000000000010
            dc.w   %0011111111111100,%0010000000000100
            dc.w   %0001111111111000,%0001111111111000
            dc.w   0,0               ; Endemarke
```

Bild 7.4: Sprite-Strukturen für einen "Pacman"

Wichtig zu wissen ist, daß Sprites immer in Lores-Auflösung (niedrige Auflösung in x- und y-Richtung) dargestellt werden, auch wenn der Viewport, auf dem sie erscheinen, eine andere Auflösung hat. Auf dem Workbench-Screen z.B. wäre ein Sprite doppelt so breit wie eine Screengrafik mit der gleichen Pixelbreite, da die Workbench in hoher x-Auflösung erscheint, der Sprite aber in niedriger.

Die Datenworte in unserem Pacman-Beispiel sind als Binärwerte abgelegt, damit man den Aufbau besser erkennen kann. Sie können sie natürlich auch in hex angeben, was zu enormer Platzersparnis führt. In der linken Spalte sehen Sie sie 16 Zeilen für die untere Plane. Alles, was hier auf 1 steht, wird in Farbe 1 (bzw. 3) angezeigt, und alles, was auf 0 steht, ist transparent. Die rechte Spalte ist die obere Plane. Die Einsen ergeben zusammen mit den Einsen der unteren Plane eine binäre 3, also wird der Punkt in Farbe 3 dargestellt. Eine Ausnahme bildet das "Auge": Hier sind die Punkte nur in der oberen Plane gesetzt, was die Farbnummer 2 ergibt. Ergebnis: Der Innenraum des Pacmans ist in Farbe 1 ausgefüllt, sein Rand hat die Farbe 3 und das Auge die Farbe 2.

So entstehen also aus den Sprite-Pixeln die Farbnummern. Jetzt müssen wir wissen, welchen Viewport-Farben diese sprite-internen Farbnummern entsprechen. Eine Amiga-Grafik kann maximal 32 frei wählbare Farben haben. Die oberen 16 werden als Spritefarben genutzt, auch wenn die angezeigte Grafik eigentlich weniger Farben hat. Jeder Sprite hat aber 3 belegbare Farben (die durchsichtige nicht mitgezählt). Da es 8 Sprites gibt, müßten 3*8, also 24 Farben zur Verfügung stehen.

Da dem nicht so ist, faßt der Amiga jeweils zwei Sprites mit aufeinanderfolgenden Nummern zu einem Paar zusammen. Dieses Paar teilt sich dann einen Farbregistersatz. Die Register-Zuordnung sieht folgendermaßen aus:

Sprites	Farbreg. 1	Farbreg. 2	Farbreg. 3
0 und 1	17	18	19
2 und 3	21	22	23
4 und 5	25	26	27
6 und 7	29	30	31

Die hier nicht vorkommenden Farbregister (16, 20, 24 und 28) enthalten die Spritefarbe Nr. 0, die aber nicht als Farbe, sondern durchsichtig dargestellt wird. Diese Farbnummern brauchen also nicht belegt zu werden. Beachten Sie, daß die Farbregister 16-31 bei Grafiken mit mehr als 4 Bitplanes sowohl für die Sprites als auch für die Bitmapgrafik verwendet werden!

Um das Aussehen eines SimpleSprites zu verändern, stellt uns Graphics die ChangeSprite-Routine zur Verfügung. Sie ändert den Zeiger auf die Grafikdaten in der SimpleSprite-Struktur und nimmt die Nötigen Anpassungen vor.

ChangeSprite = -420 (Graphics-Library)

***viewPort**	**a0**	<	Zeiger auf Viewport, in dem der zu ändernde Sprite liegt
***ssprite**	**a1**	<	Zeiger auf Struktur des zu ändernden SimpleSprites
***newdata**	**a2**	<	Zeiger auf neue Sprite-Grafikdaten
Erklärung			Ändert das Aussehen eines SimpleSprites durch Wechseln des Grafikdaten-Zeigers.

Bewegen und freigeben von SimpleSprites

Wenn Sie Ihren Sprite mit GetSprite angemeldet haben, ist er noch nicht sofort am Bildschirm zu sehen, da seine Koordinaten noch undefiniert sind. Mit der Routine MoveSprite können Sie die Koordinaten ändern. Nach der ersten Änderung wird der Sprite auch sichtbar:

MoveSprite = -426 (Graphics-Library)

***viewPort**	**a0**	<	Zeiger auf Viewport, in den der Sprite eingebunden werden soll bzw. sich befindet
***ssprite**	**a1**	<	Zeiger auf SimpleSprite-Struktur, die den Sprite verwaltet
x	**d0**	<	Gewünschte x-Koordinate
y	**d1**	<	Gewünschte y-Koordinate

Erklärung	Verschiebt einen SimpleSprite an die angegebenen Koordinaten.

Beachten Sie, daß die Sprite-Koordinaten gemäß der Auflösung des Viewports gezählt werden. Auf einem Viewport mit vertikal oder horizontal hoher Auflösung, müssen Sie einen Sprite also um zwei Punkte verschieben, damit er sich effektiv um einen Punkt bewegt. Wie Ihnen vielleicht aufgefallen ist, sprechen wir immer vom Viewport, auf dem der Sprite auftaucht, nicht vom Screen o.ä.. Sprites können sowohl auf Intuition-Screens (die auch einen Viewport haben) als auch auf "selbstgestrickten" Viewports (siehe letzter Abschnitt) erscheinen, weshalb wir die allgemeinere Bezeichnung Viewport verwenden.

Wir sehen, daß die Sprites Viewport-abhängig sind. Daraus folgt, daß sich ein Sprite immer mit dem Screen, auf dem er zu sehen ist, mitbewegt. Wir werden gleich einen Gel-Typ kennenlernen, bei dem dies nicht so ist.

Falls sich mehrere Sprites überlagern, gilt die von der Hardware festgesetzte Prioritätenfolge: Ein Sprite mit einer niedrigeren Nummer erscheint immer vor einem Sprite mit einer höheren Nummer.

Wenn Sie einen Sprite nicht mehr benötigen, melden Sie ihn mit FreeSprite ab. Er wird dadurch automatisch vom Viewport entfernt.

FreeSprite	=	**-414 (Graphics-Library)**

spriteNum	**d0**	<	Nummer des freizugebenden Sprites
Erklärung			Gibt einen SimpleSprite frei. Er wird vom Darstellungsviewport gelöscht und ist für andere Programme wieder verfügbar.

Das erste Demoprogramm meldet einen SimpleSprite (den Pacman) auf einem eigenen Screen an, der immer hinter der Maus hergesteuert wird. Je weiter die Maus vom Pacman entfernt ist, desto schneller bewegt er sich auf sie zu. Außerdem wird das Aussehen des Sprites geändert, je nachdem, ob er sich rechts oder links von der Maus befindet.

```
* Programm 7.15 (Auszug): Demo SimpleSprites

        ...
```

```
        move.l  gfxbase,a6

        move.l  vport,a0        ; Farbe 21 (Sprite 2, Farbe 1)
        moveq   #21,d0          ; einstellen
        moveq   #15,d1
        move.w  d1,d2
        move.w  d2,d3
        jsr     SetRGB4(a6)

        move.l  vport,a0        ; Farbe 22 (Sprite 2, Farbe 2)
        moveq   #22,d0          ; einstellen
        clr.w   d1
        clr.w   d2
        clr.w   d3
        jsr     SetRGB4(a6)

        move.l  vport,a0        ; Farbe 23 (Sprite 2, Farbe 3)
        moveq   #23,d0          ; einstellen
        moveq   #12,d1
        move.w  d1,d2
        move.w  d2,d3
        jsr     SetRGB4(a6)

        lea     ssprite,a0      ; SimpleSprite Nr. 2
        moveq   #2,d0
        jsr     GetSprite(a6)   ; Reservieren

        clr.w   d6              ; d6 und d7 enthalten die x-
        clr.w   d7              ; bzw. y-Position des Sprites

main:   move.l  gfxbase,a6
        jsr     WaitTOF(a6)     ; Auf Strahlrücklauf warten

* Sprite an neue Position bringen

        move.l  vport,a0        ; Viewport
        lea     ssprite,a1      ; Sprite-Struktur
        move.w  d6,d0           ; Koordinaten
        move.w  d7,d1
        jsr     MoveSprite(a6) ; Sprite bewegen

* Maus-Koordinaten feststellen

        move.l  intbase,a0      ; Basis Int-Lib
        clr.l   d0
        clr.l   d1
        move.w  $46(a0),d4      ; Mauskoordinate aus Intbase-
        move.w  $44(a0),d5      ; struktur auslesen

* Sprite an Mausbewegungen anpassen

        move.w  d4,d0           ; Wenn d6 größer oder gleich
        sub.w   #2,d0           ; d4 minus 2 ist, springe zu
        cmp.w   d0,d6           ; main2
        bge     main2
```

```
        move.l  vport,a0
        lea     ssprite,a1
        lea     image1,a2       ; Image 'Pacman rechts'
        jsr     ChangeSprite(a6)      ; Spriteaussehen ändern

        move.w  d4,d0           ; Wenn d4 minus d6 größer oder
        sub.w   d6,d0           ; gleich 15 ist, springe zu
        cmp.w   #15,d0          ; main1
        bge     main1

        add.w   #2,d6           ; Erhöhe x-Koordinate um 2
        bra     main2

main1:  move.w  d4,d0           ; Erhöhe x-Koordinate um
        sub.w   d6,d0           ; (d4 minus d6) durch 15 plus 1
        divu    #15,d0
        add.w   #1,d0
        add.w   d0,d6

main2:  move.w  d4,d0           ; Wenn d6 kleiner oder gleich
        add.w   #2,d0           ; d4 plus 2 ist, springe zu
        cmp.w   d0,d6           ; main4
        ble     main4

        move.l  vport,a0
        lea     ssprite,a1
        lea     image2,a2       ; Image 'Pacman links'
        jsr     ChangeSprite(a6)      ; Spriteaussehen ändern

        move.w  d6,d0           ; Differenz zwischen Sprite- und
        sub.w   d4,d0           ; Mausposition größer gleich 15?
        cmp.w   #15,d0
        bge     main3           ; Wenn ja, zu main3

        sub.w   #2,d6           ; Maus-x minus 2
        bra     main4

main3:  move.w  d6,d0           ; Erniedrige Maus-x um (Sprite-x
        sub.w   d4,d0           ; minus Maus-x) durch 15 plus 1
        divu    #15,d0          ; (damit sich der Sprite schneller
        add.w   #1,d0           ; bewegt, je weiter die Maus von
        sub.w   d0,d6           ; ihm weg ist)

main4:  move.w  d5,d0           ; Dieselben Tests der Maus- und Sprite-
        sub.w   #2,d0           ; Positionen, diesmal für die y-
        cmp.w   d0,d7           ; Koordinaten (siehe oben)
        bge     main6

        ...

        moveq   #2,d0           ; Sprite 2 freigeben
        jsr     FreeSprite(a6)

        move.l  intbase,a6

        ...
```

```
* Datenbereich

            ...

ssprite:    dc.l    image1
            dc.w    16
            dc.w    0,0,0

            section "",data_c

image1:     dc.w    0,0
            dc.w    %0001111111111000,%0001111111111000
            dc.w    %0011111111111100,%0010000000000100
            dc.w    %0111111100111110,%0100000011000010
            dc.w    %1111111100111111,%1000000011000011
            dc.w    %1111111111111100,%1000000000001100
            dc.w    %1111111111110000,%1000000000110000
            dc.w    %1111111111000000,%1000000011000000
            dc.w    %1111111100000000,%1000000100000000
            dc.w    %1111111111000000,%1000000011000000
            dc.w    %1111111111111000,%1000000000111000
            dc.w    %1111111111111111,%1000000000000111
            dc.w    %1111111111111111,%1000000000000001
            dc.w    %0111111111111110,%0100000000000010
            dc.w    %0011111111111100,%0010000000000100
            dc.w    %0001111111111000,%0001111111111000
            dc.w    0,0

image2:     dc.w    0,0
            dc.w    %0001111111111000,%0001111111111000
            dc.w    %0011111111111100,%0010000000000100
            dc.w    %0111110011111110,%0100001100000010
            dc.w    %1111110011111111,%1100001100000001
            dc.w    %0011111111111111,%0011000000000001
            dc.w    %0000111111111111,%0000110000000001
            dc.w    %0000001111111111,%0000001100000001
            dc.w    %0000000011111111,%0000000010000001
            dc.w    %0000001111111111,%0000001100000001
            dc.w    %0001111111111111,%0001110000000001
            dc.w    %1111111111111111,%1110000000000001
            dc.w    %1111111111111111,%1000000000000001
            dc.w    %0111111111111110,%0100000000000010
            dc.w    %0011111111111100,%0010000000000100
            dc.w    %0001111111111000,%0001111111111000
            dc.w    0,0
```

Programm 7.15 (Auszug)

Die "Verfolgungsjagd", die der Sprite ausführt, wird so bewerkstelligt: Die Position des Sprites wird mit den Mauskoordinaten, die aus der Intuition-Base-Struktur ermittelt werden, verglichen. Wenn sich der Sprite links von der Maus befindet, muß er nach rechts bewegt werden, ansonsten nach rechts (oder gar nicht, wenn er sich genau über der Maus be-

findet). Damit der Sprite nicht "zittert", wenn er genau über der Maus ist, setzen wir eine Toleranz von 2 Pixeln beim Test an. Die Abfrage der y-Position erfolgt analog.

Exemplarisch nun die Bearbeitung der x-Koordinaten: Wenn sich der Sprite höchstens 15 Pixel von der Maus entfernt befindet, wird die Position immer um zwei Pixel verändert. Sollte er weiter entfernt sein, wird die Positionsdifferenz durch 15 geteilt und die Mausposition um diesen Wert verändert. Das bewirkt, daß der Sprite umso schneller bewegt wird, je weiter er von der Maus entfernt ist (größere Differenz - größere Koordinatenänderung).

Die Änderung der y-Koordinaten läuft analog ab. Der Rest des Programms dürfte klar sein.

Attached-Sprites

Da die Beschränkung auf 3 Farben plus eine durchsichtige für manche Zwecke nicht ausreicht, stellt das System die Möglichkeit zur Verfügung, zwei "benachbarte" Sprites (mit aufeinanderfolgenden Nummern) zu einem Sprite zusammenzufügen. Ein solcher Doppelsprite nennt sich "Attached-Sprite". Zwei Sprites haben zusammen 4 Bitplanes, woraus sich eine Farbanzahl von $2^4 = 16$ ergibt. Für diese 16 Farben werden die Farbregister 16-31, also alle normalen Spritefarben, genutzt. Auch hier gibt es eine transparente Farbe: Die Stellen, an denen die Bits in beiden Planes beider Sprites auf 0 stehen, erscheinen durchsichtig. Die Farbe 16, die von der Nummern-Berechnung her für diese Farbe eigentlich benutzt würde, braucht also nicht belegt zu werden.

Es können nur jeweils die zwei Sprites zu einem Attached-Paar kombiniert werden, die sich im Normalmodus einen Farbregistersatz teilen, also die Sprites 0 und 1, 2 und 3, usw. Die Reihenfolge der Bitplanes ist folgende: Die beiden Planes des Sprites mit der niedrigeren Nummer gelten als Planes 1 und 2 des Attached-Sprites, die des anderen als Planes 3 und 4.

Um aus einem Sprite-Paar einen Attached-Sprite zu machen, muß beim Sprite mit der ungeraden Nummer (also den zweiten der beiden Sprites) in das zweite Wort der Grafikdaten (Positions-Kontrolle des Systems), in das normalerweise eine 0 geschrieben wird, eine 128 eingetragen werden. Dadurch wird das 7. Bit (von 0 an gezählt) im Wort gesetzt. Dieses Bit dient als Attached-Flag. Wichtig ist, daß die beiden Sprites, die das Attached-Paar bilden, immer an der gleichen Position stehen müssen, da sich sonst falsche Farben ergeben. Hier die Tabelle der Sprite-Planes mit den zugehörigen Farbregister-Nummern (die Sprites und Planes wieder in umgekehrter Reihenfolge):

Ungerader Sprite Plane 2	Plane 1	Gerader Sprite Plane 2	Plane 1	Farbe
0	0	0	0	durchsichtig
0	0	0	1	Register 17
0	0	1	0	Register 18
0	0	1	1	Register 19
0	1	0	0	Register 20
0	1	0	1	Register 21
0	1	1	0	Register 22
0	1	1	1	Register 23
1	0	0	0	Register 24
1	0	0	1	Register 25
1	0	1	0	Register 26
1	0	1	1	Register 27
1	1	0	0	Register 28
1	1	0	1	Register 29
1	1	1	0	Register 30
1	1	1	1	Register 31

Weil die Attached-Sprites so schön bunt aussehen, schauen wir uns zu diesem Thema auch ein Beispielprogramm an. Wir ändern das letzte Programm einfach dahingehend ab, daß wir zwei SimpleSprite-Strukturen einrichten und in der Hauptschleife beide Sprites an die selbe Position bringen. Vom Aussehen her sollen die Sprites diesmal aus zwei Farbverläufen bestehen.

```
* Programm 7.16 (Auszug): SimpleSprites als Attached-Sprites

        ...

        move.l  vport,a0        ; 32 Farben des neuen Screens
        lea     cols,a1         ; per LoadRGB4 einstellen
        moveq   #32,d0
        jsr     LoadRGB4(a6)

        lea     ssprite1,a0     ; Reserviere ersten Sprite
        moveq   #2,d0
        jsr     GetSprite(a6)

        lea     ssprite2,a0     ; Reserviere zweiten Sprite
        moveq   #3,d0
        jsr     GetSprite(a6)

        clr.w   d6              ; Wieder die x- und y-
        clr.w   d7              ; Koordinaten

main:   move.l  gfxbase,a6      ; Warte auf Strahlrücklauf
        jsr     WaitTOF(a6)

        move.l  vport,a0        ; Versetze Sprite 1 an die
        lea     ssprite1,a1     ; neue Position
```

```
        move.w  d6,d0
        move.w  d7,d1
        jsr     MoveSprite(a6)

        move.l  vport,a0        ; Sprite 2 an neue Position
        lea     ssprite2,a1
        move.w  d6,d0
        move.w  d7,d1
        jsr     MoveSprite(a6)

        ...

* Datenbereich

                ...

ssprite1:       dc.l    image1
                dc.w    16
                dc.w    0,0,0

ssprite2:       dc.l    image2
                dc.w    16
                dc.w    0,0,0

cols:           dc.w    0,$ccc,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                dc.w    $200,$400,$600,$800,$a00,$c00,$e00,$f00
                dc.w    $f0,$e0,$c0,$a0,$80,$60,$40,$20

                section "",data_c

image1:         dc.w    128,128
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %1111111111111111,%0000000000000000
                dc.w    %0000000000000000,%1111111111111111
                dc.w    %1111111111111111,%1111111111111111
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %1111111111111111,%0000000000000000
                dc.w    %0000000000000000,%1111111111111111
                dc.w    %1111111111111111,%1111111111111111
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %1111111111111111,%0000000000000000
                dc.w    %0000000000000000,%1111111111111111
                dc.w    %1111111111111111,%1111111111111111
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %1111111111111111,%0000000000000000
                dc.w    %0000000000000000,%1111111111111111
                dc.w    %1111111111111111,%1111111111111111
                dc.w    0,0

image2:         dc.w    128,128
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %0000000000000000,%0000000000000000
                dc.w    %1111111111111111,%0000000000000000
                dc.w    %1111111111111111,%0000000000000000
```

```
        dc.w    %1111111111111111,%0000000000000000
        dc.w    %1111111111111111,%0000000000000000
        dc.w    %0000000000000000,%1111111111111111
        dc.w    %0000000000000000,%1111111111111111
        dc.w    %0000000000000000,%1111111111111111
        dc.w    %0000000000000000,%1111111111111111
        dc.w    %1111111111111111,%1111111111111111
        dc.w    %1111111111111111,%1111111111111111
        dc.w    %1111111111111111,%1111111111111111
        dc.w    %1111111111111111,%1111111111111111
        dc.w    0,0
```

Programm 7.16 (Auszug)

Das Programm dürfte, da es größtenteils dem vorigen entspricht, gut verständlich sein. Die zwei Spritemuster sind so gewählt, daß jede der 16 Farben (16-31), die bei einem Attached-Sprite möglich sind, vorkommen (die korrespondierenden Zeilen und Bitplanes der beiden Sprites sind "übereinanderzulegen").

7.8.2 Die virtuellen Sprites (VSprites)

Eine weitere, unter Umständen ärgerliche Beschränkung der Sprites ist die Tatsache, daß nur 8 verschiedene Sprites zur Verfügung stehen, oft benötigt man jedoch mehr. Aufgrund der Hardware-Architektur des Amiga ist es möglich, ein und den selben Sprite mehrfach darzustellen, jedoch nur in vertikaler Richtung. Ein Sprite kann also z.B. ein paar Zeilen unter seiner Endkoordinate noch einmal angezeigt werden, sogar mit einem anderen Aussehen, jedoch nicht mehrfach im gleichen Zeilenbereich (also nebeneinander). Mindestens eine Zeile Abstand muß vom Ende der ersten Darstellung zum Anfang der zweiten eingehalten werden.

Die Berechnung, der zur mehrfachen Anzeige eines Sprites nötigen Steuerstrukturen (das sind die schon erwähnten Copperlisten), wäre eine recht komplizierte Sache, aber es gibt zum Glück eine Einrichtung in der Graphics-Library, die uns diese Arbeit abnimmt: die virtuellen (scheinbaren) Sprites, abgekürzt VSprites. Von diesen Sprites können mit gewissen Einschränkungen beliebig viele gleichzeitig auf dem Bildschirm zu sehen sein. Sogar eigene Farben kann jeder dieser virtuellen Sprites haben. Die Einschränkung betrifft die Anzahl Sprites pro Zeile (bzw. pro Zeilenbereich): Hier gilt wieder die Grenze 8. Das Grafiksystem sucht sich bei der Darstellung der VSprites selbst die günstigste Kombination der Hardware-Sprites aus.

Die VSprites sind sicherlich eine feine Sache, aber zur ihrer Benutzung müssen wir diesmal etwas mehr an Vorarbeit leisten. Wir müssen uns mit diversen Strukturen herumschlagen. Die erste, genannt GelsInfo, wurde bei der Be-

sprechung der Rastport-Struktur schon einmal kurz erwähnt. Sie dient, ähnlich wie die View-Struktur, als Haupt-Verwaltungsstruktur aller darzustellenden Gels (wozu auch die VSprites zählen), die ihrerseits untereinander verkettet sind.

Die GelsInfo-Struktur

```
00    dc.b   gi_sprRsrvd          ; Zu benutzende Hardware-Sprites
01    dc.b   gi_Flags
02    dc.l  *gi_gelHead           ; Erstes Gel
06    dc.l  *gi_gelTail           ; Letztes Gel
10    dc.l  *gi_nextLine          ; System-Zeiger
14    dc.l  *gi_lastColor         ; System-Zeiger
18    dc.l  *gi_collHandler       ; Zeiger auf Sprungtabelle für Koll.
22    dc.w   gi_leftmost          ; Linker Rand
24    dc.w   gi_rightmost         ; Rechter Rand
26    dc.w   gi_topmost           ; Oberer Rand
28    dc.w   gi_bottommost        ; Unterer Rand
30    dc.l  *gi_firstBlissObj
34    dc.l  *gi_lastBlissObj
38           gi_SIZEOF
```

gi_sprRsrvd
Dieses Byte gibt an, welche der 8 Hardware-Sprites für die Darstellung der VSprites genutzt werden sollen. Jedem Sprite wird die entsprechende Bitnummer zugeordnet. Sollen alle Sprites benutzt werden, muß hier eine 255 (alle Bits gesetzt) stehen.

***gi_gelHead, *gi_gelTail**
Diese Zeiger weisen auf den Beginn bzw. das Ende der Gel-Liste. Die einzelnen Gels werden in VSprite-Strukturen verwaltet. Auf solche Strukturen zeigen auch gelHead und gelTail. Sie enthalten das erste bzw. letzte Gel in der Liste. Als User müssen wir lediglich genügend leeren Speicher für die beiden Strukturen reservieren.

***gi_nextLine, *gi_lastColor**
Hierbei handelt es sich um zwei System-Zeiger. Als braver Benutzer brauchen Sie diese nur auf ausreichend reservierte, leere Speicherbereiche zu stellen. NextLine benötigt 16 Bytes und LastColor 32.

***gi_collHandler**
Zeigt auf eine Tabelle aus 16 Routineneinsprüngen, die bei bestimmten Kollisionen angesprungen werden (kommt noch).

gi_leftmost - gi_bottommost
Hier müssen die Grenzkoordinaten angegeben werden, innerhalb denen die VSprites angezeigt werden. Eine Überschreitung dieser Grenzen kann über eine Kollisionsroutine behandelt werden.

Für diese Struktur gibt es eine Initialisierungsroutine, aber Sie schreiben diese am besten trotzdem über 'dc'-Befehle in Ihr Programm, da einige Werte wie die nextLine- und lastColor-Zeiger und die Grenzkoordinaten sowieso "von Hand" eingetragen werden müssen. Die Routine InitGels übernimmt dann die restliche Einrichtungsarbeit:

InitGels	=	**-120 (Graphics-Library)**

***gelHead**	**a0**	<	Dummy-Zeiger auf leere VSprite-Struktur
***gelTail**	**a1**	<	Siehe a1
***gelsInfo**	**a2**	<	Zeiger auf zu initialisierende GelsInfo-Struktur

Erklärung Richtet die angegebene GelsInfo-Struktur ein und trägt unter anderem die Zeiger gelHead und gelTail darin ein.

Eine Beispiel-GelsInfo-Struktur könnte so aussehen:

```
gelsinfo:   dc.b   255           ; Alle Hardware-Sprites benutzen
            dc.b   0             ; Interne Flags
            dc.l   0,0           ; Head und Tail
            dc.l   nline         ; Platz für NextLine
            dc.l   lcolor        ; Platz für LastColor
            dc.l   0             ; Keine Kollisionstabelle
            dc.w   0,640,0,256   ; Begrenzungs-Koordinaten

nline:      ds.b   16
lcolor:     ds.b   32
```

Bild 7.5: Beispiel für eine GelsInfo-Struktur

Auf die so eingerichtete GelsInfo-Struktur muß nun noch ein Zeiger in den Rastport, in dem die Gels erscheinen sollen, geschrieben werden, und zwar in den Eintrag rp_GelsInfo, Offset 20.

Jeder VSprite wird in einer gleichnamigen Struktur verwaltet. Neben den VSprites gibt es noch einen weiteren Gel-Typ, die Bobs, zu denen wir später kommen werden. Zur Verwaltung der Bobs wird die gleiche Struktur, wie für die VSprites benutzt, weshalb einige Einträge für VSprites unwichtig sind (durch '---' markiert). Hier die Struktur:

Die VSprite-Struktur (für VSprites)

```
00   dc.l   *vs_NextVSprite    ; Zeiger auf den nächsten VSprite
04   dc.l   *vs_PrevVSprite    ; Zeiger auf den letzten VSprite
08   dc.l   *vs_DrawPath       ; ---
12   dc.l   *vs_ClearPath      ; ---
16   dc.w    vs_Oldy           ; ---
```

```
18      dc.w     vs_Oldx           ; ---
20      dc.w     vs_VSFlags        ; VSprite-Flags
22      dc.w     vs_Y              ; y-Koordinate
24      dc.w     vs_X              ; x-Koordinate
26      dc.w     vs_Height         ; Höhe in Pixeln
28      dc.w     vs_Width          ; Breite in Words (immer 1)
30      dc.w     vs_Depth          ; Tiefe (immer 2)
32      dc.w     vs_MeMask         ; Kollisionsmaske 1
34      dc.w     vs_HitMask        ; Kollisionsmaske 2
36      dc.l    *vs_ImageData      ; Zeiger auf Grafikdaten
40      dc.l    *vs_BorderLine     ; Zeiger auf BorderLine-Wort
44      dc.l    *vs_CollMask       ; Zeiger auf CollMask-Plane
48      dc.l    *vs_SprColors      ; Zeiger auf Wortfeld für Farben
52      dc.l    *vs_Bob            ; ---
56      dc.b     vs_PlanePick      ; ---
57      dc.b     vs_PlaneOnOff     ; ---
58               vs_SIZEOF
```

***vs_NextVSprite, *vs_PrevVSprite**
Diese Zeiger dienen der Verkettung der einzelnen VSprites. Sie werden (zum Glück) vom System automatisch verwaltet, d.h. die Strukturen werden gemäß der y- und x-Koordinaten aufsteigend sortiert.

vs_Flags
Folgende Flags sind möglich:

VSprite-Flag	Wert	Bedeutung
VSF_VSPRITE	$001	Gesetzt für VSprite, gelöscht für Bob
VSF_MUSTDRAW	$008	VSprite muß gezeichnet werden
VSF_GELGONE	$400	VSprite ragt über Begrenzung hinaus
VSF_OVERFLOW	$800	Zu viele VSprites in einer Zeile

VSF_VSPRITE Dieses Bit gibt an, ob in der VSprite-Struktur ein VSprite oder ein Bob beschrieben wird. Für einen VSprite muß es gesetzt sein.

VSF_MUSTDRAW Falls zu viele VSprites in einer Zeile stehen (Flag VSF_OVERFLOW gesetzt), können nicht alle dargestellt werden. Es werden dann vornehmlich die VSprites, bei denen das VSF_MUSTDRAW-Flag gesetzt ist, gezeichnet.

VSF_GELGONE Dieses Flag wird nur vom System gesetzt (der User sollte es nicht verändern), und zwar dann, wenn ein VSprite über die Begrenzungen (in der GelsInfo-Struktur angegeben) hinausragt.

VSF_OVERFLOW Ein weiteres System-Flag, das gesetzt wird, wenn sich zu viele VSprites in einer Zeile befinden.

vs_Y, vs_X
Die Koordinaten eines VSprite werden nicht über eine Graphics-Routine gesetzt, sondern durch Änderung dieser beiden Einträge in der VSprite-Struktur.

vs_Height
Die Höhe des VSprite in Pixeln.

vs_Width, vs_Depth
Diese Einträge sind nur bei Bobs variabel. Bei VSprites muß hier eine 1 (für Wort-Breite) bzw. eine 2 (Tiefe = 2 Planes) stehen.

vs_MeMask, vs_HitMask
Diese Einträge sind für die Kollisionsabfrage zwischen Gels interessant. Wir werden uns im entsprechenden Abschnitt näher damit beschäftigen. Falls Sie keine Kollisionen registrieren wollen, tragen Sie hier zwei Nullen ein.

***vs_ImageData**
Hier steht ein Zeiger auf die Grafik-Daten des VSprites. Diese sind fast genauso aufgebaut wie die Daten eines SimpleSprite. Es fehlen lediglich die Null-Worte für die Positionskontrolle (vor den Grafikdaten) und die Ende-Kennung (hinter den Daten).

***vs_BorderLine, *vs_CollMask**
Siehe vs_MeMask, vs_HitMask

***vs_SpriteColors**
Dieser Zeiger weist auf ein Feld aus vier Worten, die die Farben des VSprites angeben, wobei die erste Farbe unwichtig, weil transparent ist. Hier gilt nicht die Einschränkung der SimpleSprites, daß sich je zwei Sprites einen Farbsatz teilen.

Nun ein Beispiel für eine VSprite-Struktur. Wir verwenden wieder die Pacman-Grafik aus dem vorletzten Demoprogramm:

```
vsprite:    dc.l    0,0             ; Verkettungszeiger
            dc.l    0,0             ; Nur für Bobs
            dc.w    0,0             ; Ebenfalls
            dc.w    1               ; Flags: VSPRITE, MUSTDRAW
            dc.w    20,20           ; Start-Koordinaten
            dc.w    15,1,2          ; Höhe, Breite, Tiefe
            dc.w    0,0             ; Kollisionsmasken
            dc.l    sprdata         ; Zeiger auf Grafikdaten
            dc.l    0,0             ; BorderLine und CollMask
            dc.l    sprcol          ; Zeiger auf Farbfeld
            dc.l    0               ; Kein Bob-Zeiger
            dc.b    0,0             ; Nur für Bobs

            section "",data_c       ; Folgende Daten ins Chip-RAM

sprdata:    dc.w    %0001111111111000,%0001111111111000
            dc.w    %0011111111111100,%0010000000000100
            dc.w    %0111111100111110,%0100000011000010
            dc.w    %1111111100111111,%1000000011000011
            dc.w    %1111111111111100,%1000000000001100
```

```
            dc.w    %1111111111110000,%1000000000110000
            dc.w    %1111111111000000,%1000000011000000
            dc.w    %1111111100000000,%1000000100000000
            dc.w    %1111111111000000,%1000000011000000
            dc.w    %1111111111111000,%1000000000111000
            dc.w    %1111111111111111,%1000000000000111
            dc.w    %1111111111111111,%1000000000000001
            dc.w    %0111111111111110,%0100000000000010
            dc.w    %0011111111111100,%0010000000000100
            dc.w    %0001111111111000,%0001111111111000

sprcol:     dc.w    0,$fff,0,$ccc        ; Farben: Weiß, Schwarz, Grau
```

Bild 7.6: Beispiel für eine VSprite-Struktur

Anzeigen, Bewegen und Entfernen von VSprites

Hat man die VSprite-Struktur fertig aufgebaut, kann man den VSprite mit folgender Routine in die Gels-Liste einhängen:

AddVSprite	=	**-102 (Graphics-Library)**

***VSprite**	**a0**	<	Zeiger auf Struktur des einzubindenden VSprites
***rastPort**	**a1**	<	Zeiger auf Rastport, in den der VSprite eingebunden werden soll
Erklärung			Bindet einen VSprite in die Gels-Liste eines Rastports ein.

Wie bei den SimpleSprites gilt auch hier, daß ein VSprite nach Aufruf von AddVSprite noch nicht sofort auf dem Bildschirm zu sehen ist. Vorher müssen noch ein paar Routinen aufgerufen werden:

SortGList	=	**-150 (Graphics-Library)**

***rastPort**	**a1**	<	Zeiger auf Rastport, dessen Gels-Liste sortiert werden soll
Erklärung			Sortiert die VSprite-Verkettungen anhand der eventuell geänderten x- und y-Koordinaten.

DrawGList	=	**-114 (Graphics-Library)**

***viewPort** a0 < Zeiger auf Viewport, in dem die Gels erscheinen sollen

***rastPort** a1 < Zeiger auf Rastport, aus dem die Gels stammen

Erklärung Erstellt die zur Darstellung der Gels notwendige Copperliste im angegebenen Viewport (Viewport-Eintrag vp_SprIns).

Nun müssen noch die beiden aus dem Viewport-Abschnitt schon bekannten Routinen **MrgCop** und **LoadView** aufgerufen werden, daraufhin sind die VSprites am Bildschirm zu sehen. Durch MrgCop wird die neu erstellte Sprite-Copperliste, zu finden im Eintrag vp_SprIns des Viewports, in die Gesamtcopperliste des View übernommen. VSprites sind daher View- und nicht Viewport-abhängig. Das macht sich dadurch bemerkbar, daß sie ihre Position nicht verändern, wenn man den Screen herunterzieht.

Die Positionsänderung von VSprites geschieht durch Änderung der Einträge vs_Y und vs_X in der VSprite-Struktur und anschließendem erneutem Aufruf der gerade beschriebenen vier Routinen.

Man kann einen VSprite mit folgender Routine vom Bildscchnirm und aus der Gels-Liste entfernen:

RemVSprite	=	**-138 (Graphics-Library)**

***vSprite** a0 < Zeiger auf Struktur des zu entfernenden VSprites

Erklärung Entfernt den angegebenen VSprite vom Bildschirm und aus der Gels-Liste des Rastports.

Das Beispielprogramm zu diesem Thema verwendet wieder die schon bekannte Pacman-Grafik. Es läßt zwei solche VSprites über den Bildschirm ziehen, einen waagerecht, einen senkrecht. Dabei werden die Positionen abgefragt und die Bewegungsrichtung umgekehrt, wenn die Sprites den Bildschirmrand erreichen.

```
* Programm 7.17 (Auszug): Demo VSprites

        ...
```

```
* GelsInfo und VSprites einrichten

        lea     ghead,a0        ; Dummy-Listenkopf
        lea     gtail,a1        ; Dummy-Listenende
        lea     ginfo,a2        ; Zeiger auf GelsInfo
        jsr     InitGels(a6)    ; GelsInfo einrichten

        lea     ginfo,a0        ; GelsInfo-Zeiger in
        move.l  a0,20(a4)       ; Rastport eintragen

        lea     vspr1,a0        ; Bearbeite VSprite #1

        move.w  #9,20(a0)       ; Flags
        move.w  #15,26(a0)      ; Höhe
        move.w  #128,22(a0)     ; x-Startposition
        move.w  #0,24(a0)       ; y-Startposition
        move.w  #1,28(a0)       ; Breite in Worten
        move.w  #2,30(a0)       ; Anzahl Planes
        move.l  #image,36(a0)   ; Zeiger auf Grafikdaten
        move.l  #col1,48(a0)    ; Zeiger auf Farben

        move.l  a4,a1           ; Zeiger auf Rastport
        jsr     AddVSprite(a6)  ; VSprite einbinden

        lea     vspr2,a0        ; Die selbe Prozedur für VSprite #2

        move.w  #9,20(a0)
        move.w  #15,26(a0)
        move.w  #0,22(a0)       ; x-Start
        move.w  #160,24(a0)     ; y-Start
        move.w  #1,28(a0)
        move.w  #2,30(a0)
        move.l  #image,36(a0)
        move.l  #col1,48(a0)

        move.l  a4,a1
        jsr     AddVSprite(a6)

* Hauptschleife: VSprites bewegen

main2:  move.l  gfxbase,a6
        jsr     WaitTOF(a6)     ; Auf Strahlrücklauf warten

        lea     vspr1,a0        ; Bearbeite VSprite 1
        move.w  dir1,d0         ; Addiere 'dir1' zur x-Pos.
        add.w   d0,24(a0)
        cmp.w   #320,24(a0)     ; x-Pos. größer als 320?
        bgt     main3           ; Wenn ja
        cmp.w   #0,24(a0)       ; Oder kleiner als 0?
        blt     main3           ; Wenn ja
        bra     main4
main3:  neg.w   dir1            ; Bewegungsrichtung umkehren

main4:  lea     vspr2,a0        ; Bearbeite VSprite 2
        move.w  dir2,d0         ; Addiere 'dir2' zur y-Pos.
        add.w   d0,22(a0)
```

```
        cmp.w    #255,22(a0)     ; y-Pos. größer als 255?
        bgt      main5
        cmp.w    #0,22(a0)       ; Oder kleiner als 0?
        blt      main5
        bra      main6
main5:  neg.w    dir2            ; Bewegungsrichtung umkehren

main6:  move.l   a4,a1           ; Gels-Liste sortieren
        jsr      SortGList(a6)
        move.l   vport,a0        ; Copperlisten berechnen
        move.l   a4,a1
        jsr      DrawGList(a6)
        move.l   view,a1         ; Copperlisten zusammenfügen
        jsr      MrgCop(a6)
        move.l   view,a1         ; Copperlisten neu darstellen
        jsr      LoadView(a6)

        ...

* Datenbereich

              ...

ginfo:        dc.b     255
              dc.b     0
              dc.l     0,0
              dc.l     nline
              dc.l     lcol
              dc.l     ctab
              dc.w     0,320,0,256
              dc.l     0,0

ghead:        dcb.b    58,0
gtail:        dcb.b    58,0
ctab:         dcb.b    64,0

vspr1:        dcb.b    58,0
vspr2:        dcb.b    58,0

dir1:         dc.w     3
dir2          dc.w     3

col1:         dc.w     $fff,$000,$ccc

              section  "",data_c

nline:        dcb.b    16,0
lcol:         dcb.b    32,0

image:        dc.w     %0001111111111000,%0001111111111000
              ...
```

Programm 7.17 (Auszug)

7.8.3 Die Bobs (Blitter Objects)

Die SimpleSprites und VSprites wurden von einer speziellen Hardware-Einrichtung, unabhängig von der eigentlichen Bildschirmgrafik, dargestellt oder in die sonstige Grafik eingemischt. Der Gel-Typ, mit dem wir uns jetzt beschäftigen wollen, wird auf eine ganz andere Weise erzeugt.

Die Abkürzung 'Bob' steht für 'Blitter Object', woraus man schon ersehen kann, daß der Coprozessor Blitter für sie verantwortlich ist. Dieser Coprozessor ist uns schon einmal im Zusammenhang mit den Grafik-Kopierroutinen begegnet. Wir haben erfahren, daß er in der Lage ist, sehr schnell rechteckig-orientierte Speicherbereiche (sprich Grafiken) zu verschieben. Diese Tatsache nutzt die Graphics-Library aus und stellt uns Gels zur Verfügung, die nicht unabhängig von der Bitmap-Grafik laufen, sondern vom Blitter in sie hineinkopiert werden.

Diese Verfahrensweise hat sowohl Vor- als auch Nachteile (wobei die Vorteile aber überwiegen). Erstens fällt die von den Sprites her bekannte Beschränkung in der Breite (16 Pixel) und der Tiefe (3 Farben plus eine durchsichtige) weg. Da die Bobs direkt in die Bitmap-Grafik einkopiert werden, ergibt sich ihre maximale Größe und Farbanzahl aus den Daten der Bitmap, in der sie erscheinen.

Der Nachteil besteht darin, daß die Bitmap-Grafik an der Stelle, an der der Bob erscheint, vorher gesichert werden muß, da Bobs die Grafik (im Gegensatz zu Sprites) wirklich überschreiben. Nach einem bißchen Vorarbeit übernimmt das allerdings das System für uns. Außerdem sind Bobs, vor allem, wenn sie besonders groß und/oder vielfarbig sind, langsamer in der Darstellung als Sprites.

Das Zeichnen eines Bobs besteht im wesentlichen aus zwei Teilen: dem Löschen des Bobs an der alten Position durch Zurückkopieren der gesicherten Grafik und dem Zeichnen an der neuen Position. Eingeleitet wird das Zeichnen mit den schon bekannten Routinen **SortGList** und **DrawGList**.

Zur Benutzung von Bobs muß, wie schon bei den VSprites, die GelsInfo-Struktur initialisiert und der Zeiger auf sie in rp_GelsInfo eingetragen sein. Die Bobs selbst werden mit der gleichen Struktur verwaltet wie die VSprites, obwohl sie ganz anderer Natur sind. Ein paar Einträge, die bei VSprites keine Bedeutung hatten, werden jetzt allerdings benötigt, während andere nicht mehr verwendet werden. Nun die VSprite-Struktur für Bobs. Die Felder, die hier nicht kommentiert sind, haben für Bobs die selbe Bedeutung wie für VSprites.

Die VSprite-Struktur (für Bobs)

```
00      dc.l    *vs_NextVSprite         ; Zeiger auf den nächsten Bob
04      dc.l    *vs_PrevVSprite         ; Zeiger auf den letzten Bob
08      dc.l    *vs_DrawPath            ; Reihenfolge beim Zeichnen
```

```
12      dc.l   *vs_ClearPath       ; Reihenfolge beim Löschen
16      dc.w    vs_Oldy            ; Vorige y-Koordinate
18      dc.w    vs_Oldx            ; Vorige x-Koordinate
20      dc.w    vs_VSFlags         ; VSprite-Flags
22      dc.w    vs_Y               ; y-Koordinate
24      dc.w    vs_X               ; x-Koordinate
26      dc.w    vs_Height          ; Höhe in Pixeln
28      dc.w    vs_Width           ; Breite in Words
30      dc.w    vs_Depth           ; Tiefe
32      dc.w    vs_MeMask          ; Kollisionsmaske 1
34      dc.w    vs_HitMask         ; Kollisionsmaske 2
36      dc.l   *vs_ImageData       ; Zeiger auf Grafikdaten
40      dc.l   *vs_BorderLine      ; Zeiger auf BorderLine-Wort
44      dc.l   *vs_CollMask        ; Zeiger auf CollMask-Plane
48      dc.l   *vs_SprColors       ; ---
52      dc.l   *vs_Bob             ; Zeiger auf Bob-Struktur
56      dc.b    vs_PlanePick       ; Maske für benutze Planes
57      dc.b    vs_PlaneOnOff      ; Plane-Maske zum Ein/Ausschalten
58              vs_SIZEOF
```

***vs_DrawPath, *vs_ClearPath**
Diese Zeiger dienen dem System zum Merken der Reihenfolge, in der die Bobs gezeichnet bzw. gelöscht werden müssen.

vs_Flags
Folgende Flags sind für Bobs möglich:

Bob-Flag	Wert	Bedeutung
VSF_VSPRITE	$001	Gesetzt für VSprite, gelöscht für Bob
VSF_SAVEBACK	$002	Bitmap unter Bob sichern
VSF_OVERLAY	$004	Bob-Pixel in Farbe 0 durchsichtig
VSF_BACKSAVED	$100	Bitmap unter Bob wurde gesichert
VSF_BOBUPDATE	$200	System-intern
VSF_GELGONE	$400	VSprite ragt über Begrenzung hinaus

VSF_VSPRITE Im Falle eines Bobs darf dieses Flag nicht gesetzt sein (keine 1 auf den Flag-Wert aufaddieren).

VSF_SAVEBACK Wenn dieses Flag gesetzt ist, sichert das System die Grafik, die vorher an den Bob-Koordinaten zu sehen war, und schreibt sie bei Bob-Bewegungen wieder zurück.

VSF_OVERLAY Dieses Flag besagt, daß die Punkte des Bobs, die die Farbe 0 haben (Bits in allen Planes gelöscht), durchsichtig erscheinen sollen (also so wie die Sprites). Ist das Flag gelöscht, werden die 0-Farb-Punkte des Bobs wirklich in der Hintergrundfarbe gezeichnet.

VSF_BACKSAVED Dieses Flag wird, wie auch die folgenden, nur vom System gesetzt (der User sollte es nicht verändern), und zwar dann, wenn die Grafik unter einem Bob gesichert wurde. Das Flag dient zur Verhinderung einer Grafikrestauration, wenn der Bob zum ersten Mal dargestellt wird, und es somit noch gar keine Grafik zu restaurieren gibt.

VSF_GELGONE Dieses Flag wird gesetzt, wenn ein Bob über die Begrenzungen (GelsInfo-Struktur) hinausragt.

vs_Width, vs_Depth
Diese Einträge sind bei Bobs variabel. Die Breite wird in Worten angegeben, ein Bob muß also immer eine Breite haben, die ein Vielfaches von 16 ist. Die Tiefe kann maximal so groß sein wie die der Zielbitmap.

***vs_ImageData**
Hier steht ein Zeiger auf die Grafik-Daten des Bob. Mit ihnen werden wir uns gleich beschäftigen.

***vs_BorderLine, *vs_CollMask**
Bei Bobs müssen diese beiden Zeiger auch dann initialisiert werden, wenn man nicht mit Kollisionsabfragen arbeiten will, da die CollMask hier quasi als "Schatten-Plane" des Bobs benutzt wird. Im Anschluß an diese Struktur werden wir uns damit beschäftigen.

***vs_Bob**
Ein Zeiger auf eine Bob-Struktur, die weitere Informationen über den Bob enthält (kommt im Anschluß an diese Struktur).

vs_PlanePick, vs_PlaneOnOff
Dienen der Angabe, welche Planes der Bitmap in den Bob-Daten vorhanden sind und wie die übrigen behandelt werden sollen. Kommt auch gleich.

Bevor wir den Aufbau der Bob-Grafikdaten behandeln, erläutern wir Ihnen zunächst die Bob-Struktur, auf die ein Zeiger in der VSprite-Struktur erwartet wird:

Die Bob-Struktur

```
00    dc.w     bob_BobFlags       ; Bob-Flags
02    dc.l    *bob_SaveBuffer     ; Speicher zur Bitmap-Sicherung
06    dc.l    *bob_ImageShadow    ; Zeigt auf die CollMask
10    dc.l    *bob_Before         ; Für Zeichenreihenfolge
14    dc.l    *bob_After          ; Für Zeichenreihenfolge
18    dc.l    *bob_VSprite        ; Zeiger zurück zu VSprite
22    dc.l    *bob_BobComp        ; Zeiger auf AnimComp-Struktur
26    dc.l    *bob_DBuffer        ; Zeiger für double-buffering
30             bob_SIZEOF
```

bob_BobFlags
Folgende Flags, die teilweise system-intern sind, sind möglich:

Bob-Flag	Wert	Bedeutung
BF_SAVEBOB	$0001	Hintergrund nicht restaurieren
BF_BOBISCOMP	$0002	Bob gehört zu einer Animation
BF_BWAITING	$0100	Intern

BF_BDRAWN	$0200	Intern
BF_BOBSAWAY	$0400	Bob beim nächsten Zeichnen entfernen
BF_BOBNIX	$0800	Bob wurde entfernt
BF_SAVEPRESERVE	$1000	Intern
BF_OUTSTEP	$2000	Intern

BF_SAVEBOB Dieses Flag kann gesetzt werden, wenn die Hintergrundrestaurierung vorübergehend außer Kraft gesetzt werden soll. Der Hintergrund wird dann zwar weiterhin gesichert, aber nicht mehr zurückgeschrieben. Der Bob wird somit neu gezeichnet, an der alten Position aber nicht mehr gelöscht, was den Brush-Effekt diverser Malprogramme ergibt. Falls die Restaurierung generell unterbunden werden soll, sollten Sie statt dieses Flags VSF_SAVEBACK benutzen, d.h. selbiges Flag löschen.

BF_BOBISCOMP Dieses Flag gibt an, daß der Bob zu einer Animations-Struktur gehört.

BF_BOBSAWAY Wenn dieses Flag gesetzt ist, wird der Bob beim nächsten Zeichnen mit DrawGList aus der Gels-Liste entfernt.

BF_BOBNIX Dieses Flag wird vom System als Quittung von BF_BOBSAWAY gesetzt, wenn der Bob entfernt wurde.

***bob_SaveBuffer**
Dieser Zeiger weist auf einen Speicherbereich im Chip-RAM, der groß genug sein muß, um die komplette Grafik, die vorher an den Bob-Koordinaten zu sehen war, aufzunehmen. Die Größe in Byte berechnet sich aus Wortbreite * 2 * Höhe * Planes (der Bitmap-Grafik). Der Zeiger muß nur eingerichtet werden, wenn der Bobhintergrund auch gesichert werden soll (Flag VSF_SAVEBACK gesetzt).

***bob_ImageShadow**
Hier kann der Zeiger auf den gleichen Speicherbereich wie für vs_CollMask eingetragen werden.

***bob_Before, *bob_After**
Falls Sie eine Zeichenreihenfolge für Ihre Bobs festlegen möchten, können Sie in diese beiden Zeiger die Strukturadressen der Bobs eintragen, die vor bzw. nach dem aktuellen Bob gezeichnet werden sollen. Ansonsten tragen Sie einfach Nullen ein.

***bob_VSprite**
Dieser Eintrag zeigt zurück zur VSprite-Struktur, von der aus auf die Bob-Struktur verwiesen wird (doppelte Verknüpfung der Strukturen).

***bob_Comp, *bob_DBuffer**
Diese beiden Zeiger werden nur für die Spezialfunktionen Bob-Animation bzw. Double-Buffering benötigt und sind für uns weniger interessant.

Nach dieser Struktur sollen nun all die Dinge geklärt werden, von denen es hieß 'kommt gleich'. Als erstes zum Aufbau der Bob-Grafikdaten.

Aufbau der Bob-Grafikdaten und -Masken

Sie erinnern sich vielleicht noch an den Abschnitt über die Image-Struktur im Intuition-Kapitel. Wenn ja, werden Ihnen die Begriffe PlanePick und PlaneOnOff sicherlich bekannt vorkommen. Bis auf die Tatsache, daß die Breite eines Bobs in Worten und nicht in Pixeln angegeben wird und sie daher immer ein Vielfaches von 16 betragen muß, sind die Bob-Grafikdaten identisch aufgebaut wie die Image-Grafikdaten.

Zur Erinnerung: Eine Amiga-Grafik ist in Bitplanes aufgebaut, die Farbtabellennummer jedes Punktes ergibt sich aus dem "Übereinanderlegen" der zusammengehörigen Bits aus allen Bitplanes. Im Gegensatz zu den Sprite-Grafikdaten werden bei den Bobs nicht die Bitplanes zeilenweise aufgeführt (Zeile 1 Plane 1, Zeile 1 Plane 2, Zeile 2 Plane 1, Zeile 2 Plane 2 usw.), sondern zuerst kommen alle Zeilen der ersten Plane, dann alle der zweiten Plane usw. Im PlanePick-Eintrag muß angegeben werden, welche Planes der Bob-Zielgrafik in den Bob-Grafikdaten vorhanden sind, also in die Zielgrafik kopiert werden sollen.

Setzt man z.B. die Bits 1, 3 und 5 im PlanePick-Byte (Zählung ab Bit 0), so bestehen die Bob-Grafikdaten nur aus 3 Planes. Die erste dieser Planes (Nr. 0) wird in die erste Plane, deren Bit im PlanePick-Byte gesetzt ist, kopiert. In unserem Fall also Plane 1. Die zweite Bob-Plane kommt in die Grafik-Plane 3, die dritte in Plane 5. Noch ein Beispiel: Bei Setzen der PlanePick-Bits 0, 1 und 2 werden die Bob-Planes 0, 1 und 2 auch in die Grafik-Planes 0, 1 und 2 kopiert.

Das PlaneOnOff-Byte bestimmt nun, was mit den Grafik-Planes, die nicht im PlanePick-Byte erfaßt sind (deren Bits also auf 0 stehen), geschehen soll. Ist ein PlaneOnOff-Bit gesetzt, so werden die Bits in der zugehörigen Plane auch auf 1 bzw. auf den Zustand des entsprechenden Bits in der Bob-Schattenmaske (kommt gleich) gesetzt. Bei gelöschtem PlaneOnOff-Bit wird das Plane-Bit immer entfernt.

Nun zu den Rand- und Schattenmasken. Wie schon erwähnt, werden sie gewöhnlich nur für die Kollisionserkennung benötigt, bei den Bobs aber hat die Schattenmaske noch einen anderen Zweck: Sie wird in all die Grafik-Planes kopiert, deren PlanePick-Bit gelöscht und deren PlaneOnOff-Bit gesetzt ist.

Für die Randmaske muß soviel Speicher im Chip-RAM bereitstehen, daß eine Zeile des Bobs hineinpaßt. Berechnet wird die Randmaske (das gilt auch für die Sprites), indem alle Zeilen aller Planes des Bobs OR-Verknüpft werden. Ein Punkt in der Randmaske ist also dann gesetzt, wenn in irgendeiner Bob-Zeile (oder auch in mehreren) an der entsprechenden Stelle

ein gesetzter Punkt ist. Mit der Schattenmaske verhält es sich ähnlich: Sie hat die Größe eines Bob-Planes, in Bytes also Breite * 2 * Höhe. Sie entsteht durch OR-Verknüpfung aller Bob-Planes.

Glücklicherweise muß die Berechnung der Masken nicht von Hand durchgeführt werden. Es gibt dafür eine Graphics-Routine. Sie müssen lediglich den Speicher (im Chip-RAM!) bereitstellen und die Zeiger vs_BorderLine und vs_CollMask setzen.

InitMasks	**=**	**-126 (Graphics-Library)**

***VSprite**	**a0**	<	Zeiger auf VSprite (bzw. Bob), dessen Masken initialisiert werden sollen.
Erklärung			Berechnet die BorderLine- und CollMask-Masken eines VSprites bzw. Bobs. Die entsprechenden Zeiger in der VSprite-Struktur müssen gesetzt sein.

Anzeigen, Bewegen und Entfernen von Bobs

Nachdem die VSprite- und Bob-Struktur auf diese Weise vorbereitet sind, können wir den Bob auf den Bildschirm bringen. Dazu muß er zunächst einmal in die Bob-Kette des gewünschten Rastports "eingehängt" werden, was die Routine AddBob erledigt:

AddBob	**=**	**-96 (Graphics-Library)**

***bob**	**a0**	<	Zeiger auf Bob-Struktur (nicht VSprite!) des einzuhängenden Bob
***rastPort**	**a1**	<	Zeiger auf Rastport, in dem der Bob erscheinen soll
Erklärung			Hängt einen Bob in die GelsInfo-Liste eines Rastports ein.

Wichtig ist, daß hier in a0 ein Zeiger auf die Bob-Struktur und nicht auf die VSprite-Struktur erwartet wird. VSprite- und Bob-Struktur sind durch gegenseitige Zeiger verbunden (vs_Bob bzw. bob_VSprite).

Zur endgültigen Darstellung am Bildschirm sind Aufrufe der schon von den VSprites her bekannten Routinen **SortGList** und **DrawGList** nötig. Im Gegensatz zu den VSprites reichen diese beiden Routinen aber schon aus, MrgCop und LoadView sind <u>nicht</u> nötig (klar, Bobs kommen direkt in die Bitmap-Grafik und brauchen daher keine Copperlisten, zu deren Berechnung MrgCop dient).

Für das Bewegen der Bobs gilt ebenfalls dasselbe wie für die VSprites: Man ändert die Einträge vs_Y und vs_x in der VSprite-Struktur und ruft SortGList und DrawGList erneut auf.

Entfernen kann man einen Bob auf zwei Arten. Die erste beruht auf der Routine RemIBob:

RemIBob	=	**-132 (Graphics-Library)**

***bob**	**a0**	<	Zeiger auf den zu entfernende Bob-Struktur
***rastPort**	**a1**	<	Zeiger auf Rastport, von dem der Bob entfernt werden soll
***viewPort**	**a2**	<	Zeiger auf Viewport, auf dem der Bob vorher zu sehen war

Erklärung Entfernt einen Bob aus der Gels-Liste des angegebenen Rastports und gleichzeitig vom Bildschirm, d.h. vom angegebenen Viewport.

Das 'I' in RemIBob steht für 'immediately', (sofort). Der Bob wird also nicht nur aus der Gels-Liste, sondern auch sofort vom Bildschirm entfernt.

Die zweite Methode läuft folgendermaßen: Man setze das BF_BOBSAWAY-Flag im Eintrag bob_BobFlags, und beim nächsten SortGList-DrawGList-Aufruf verschwindet der Bob vom Schirm und aus der Gels-Liste. Als "Quittung" hierfür wird das Flag BF_BOBNIX gesetzt. Welche Methode Sie wählen, bleibt Ihnen überlassen; die Methode mit dem Flag ist besonders dann sinnvoll, wenn Sie viele Bobs auf einmal entfernen wollen (Flags setzen und dann eine Library-Routine aufrufen geht schneller, als für jeden Bob eine Routine aufzurufen).

Das Demoprogramm läuft fast genauso ab wie das vorige. Nur verwenden wir diesmal - natürlich - anstatt zweier VSprites zwei Bobs, die recht "breit" sind, damit der Demonstrationseffekt (Aufhebung der Breiten-Begrenzung) auch zum Tragen kommt.

```
* Programm 7.18 (Auszug): Demo Bobs

        ...

        lea     vspr1,a0        ; Masken für Bob 1 berechnen
        jsr     InitMasks(a6)
        lea     bob1,a0         ; Bob zu Rastport hinzufügen
        move.l  a4,a1
        jsr     AddBob(a6)
```

```
        lea      vspr2,a0        ; Selbiges für Bob 2
        jsr      InitMasks(a6)
        lea      bob2,a0
        move.l   a4,a1
        jsr      AddBob(a6)

        ...

main6:  move.l   a4,a1           ; Bobs an neuer Position zeichnen
        jsr      SortGList(a6)   ; (MrgCop und LoadView sind hier
        move.l   vport,a0        ; nicht nötig)
        move.l   a4,a1
        jsr      DrawGList(a6)

        ...

* Datenbereich

              ...

vspr1:        dc.l    0,0,0,0
              dc.w    0,0
              dc.w    6,128,0,20,2,1
              dc.w    0,0
              dc.l    image
              dc.l    bline
              dc.l    cmask
              dc.l    0
              dc.l    bob1
              dc.b    1,0
bob1:         dc.w    0
              dc.l    sbuff1,cmask,0,0
              dc.l    vspr1,0,0

vspr2:        dc.l    0,0,0,0
              dc.w    0,0
              dc.w    6,0,160,20,2,1
              dc.w    0,0
              dc.l    image
              dc.l    bline
              dc.l    cmask
              dc.l    0
              dc.l    bob2
              dc.b    1,0
bob2:         dc.w    0
              dc.l    sbuff2,cmask,0,0
              dc.l    vspr2,0,0

              section "",data_c

image:        dc.l    %11111111111111111111111111111111
              dc.l    %11000000000000000000000000000011
              dc.l    %10100000000000000000000000000101
              dc.l    %10011111111111111111111111111001
              dc.l    %10010000000000011000000000001001
              dc.l    %10010000000000011000000000001001
```

```
            dc.l    %10010000000000011000000000001001
            dc.l    %10010000000000011000000000001001
            dc.l    %10010000000001111110000000001001
            dc.l    %10011111111111111111111111111001
            dc.l    %10011111111111111111111111111001
            dc.l    %10010000000001111110000000001001
            dc.l    %10010000000000011000000000001001
            dc.l    %10010000000000011000000000001001
            dc.l    %10010000000000011000000000001001
            dc.l    %10010000000000011000000000001001
            dc.l    %10011111111111111111111111111001
            dc.l    %10100000000000000000000000000101
            dc.l    %11000000000000000000000000000011
            dc.l    %11111111111111111111111111111111

sbuff1:     dcb.b   160,0
sbuff2:     dcb.b   160,0
```

Programm 7.18 (Auszug)

7.8.4 Abfrage von Gel-Kollisionen

Das Amiga-System kennt zwei grundlegende Arten von Gel-Kollisionen: Berührungen von Gels untereinander und Berührungen von Gels mit der Bitmap-Grafik. Bei den VSprites und Bobs kommt noch die Möglichkeit hinzu, ein Überschreiten der in der GelsInfo-Struktur angegebenen Grenzkoordinaten als Kollision zu registrieren.

Kollisionsabfrage bei SimpleSprites

In Bezug auf die Kollisionsabfrage machen die SimpleSprites ihrem Namen wieder alle Ehre: Die Abfrage ist recht primitiv und einfach zu realisieren, läßt aber keine so komfortable Auswahl der abzufragenden Objekte zu wie die VSprites und Bobs.

Die Graphics-Library stellt keine Routinen zur Abfrage von SimpleSprite-Kollisionen zur Verfügung (im Gegensatz zu den VSprites und Bobs), wir müssen daher auf zwei Hardware-Register zurückgreifen. Die beiden Register CLXCON und CLXDAT, die als Wort-Adressen angesprochen werden können, dienen der Einstellung (CLXCON) bzw. Abfrage (CLXDAT) der Kollisionsparameter.

CLXCON ist an der Adresse **$DFF098** zu finden und bestimmt, welche Ereignisse als Kollisionen registriert werden sollen. Es können Berührungen eines Sprites mit einem Sprite aus einem anderen Paar, nicht aber Kollisionen zweier Sprites eines Paares untereinander, registriert werden, ferner Berührungen eines Sprites mit der Bitmap-Grafik. Die einzelnen Bits des CLXCON-Registers legen die erlaubten Ereignisse fest:

Belegung des CLXCON-Registers ($DFF098)

Bit	Melde bei gelöschtem Bit Kollisionen von bzw. mit	Melde bei gesetztem Bit Kollisionen von bzw. mit
15	Sprites 6 und 7	Nur Sprite 6
14	Sprites 4 und 5	Nur Sprite 4
13	Sprites 2 und 3	Nur Sprite 3
12	Sprites 0 und 1	Nur Sprite 0
11	---	Bitplane 6
10	---	Bitplane 5
9	---	Bitplane 4
8	---	Bitplane 3
7	---	Bitplane 2
6	---	Bitplane 1
5	Gelöschtem Plane-Bit 6	Gesetztem Plane-Bit 6
4	Gelöschtem Plane-Bit 5	Gesetztem Plane-Bit 5
3	Gelöschtem Plane-Bit 4	Gesetztem Plane-Bit 4
2	Gelöschtem Plane-Bit 3	Gesetztem Plane-Bit 3
1	Gelöschtem Plane-Bit 2	Gesetztem Plane-Bit 2
0	Gelöschtem Plane-Bit 1	Gesetztem Plane-Bit 1

Wie wir sehen, werden Kollisionen der Sprites mit geraden Nummern immer registriert, für die Erkennung von Kollisionen ungerader Sprites müssen die entsprechenden Bits gesetzt werden. Die Bits 11-6 bestimmten, ob Kollisionen mit der entsprechenden Bitplane gemeldet werden sollen (Bit gesetzt). Die Bits 5-0 geben schließlich an, ob ein gesetztes oder gelöschtes Plane-Bit in den zu registrierenden Planes die Kollision auslösen soll.

Dieses und auch das CLXDAT-Register sind immer wort-weise anzusprechen ('move.w' oder ähnliche Befehle). Beachten Sie, daß das CLXCON-Register aufgrund der Hardware-Architektur nur geschrieben und CLXDAT nur gelesen werden kann!

Die eigentliche Abfrage der Kollisionen erfolgt über das Register CLXDAT (an Adresse **$DFF00E**). Jedes Bit steht für ein bestimmtes Kollisionsereignis:

Die Belegung des CLXDAT-Registers ($DFF00E)

Bit	Meldet Kollision von
15	--- (unbenutzt)
14	Sprite 4 (oder 5) mit Sprite 6 (oder 7)
13	Sprite 2 (oder 3) mit Sprite 6 (oder 7)
12	Sprite 2 (oder 3) mit Sprite 4 (oder 5)
11	Sprite 0 (oder 1) mit Sprite 6 (oder 7)
10	Sprite 0 (oder 1) mit Sprite 4 (oder 5)
9	Sprite 0 (oder 1) mit Sprite 2 (oder 3)
8	Bitplane (gerade Nummer) mit Sprite 6 (oder 7)
7	Bitplane (gerade Nummer) mit Sprite 4 (oder 5)
6	Bitplane (gerade Nummer) mit Sprite 2 (oder 3)
5	Bitplane (gerade Nummer) mit Sprite 0 (oder 1)

4	Bitplane (ungerade Nummer) mit Sprite 6 (oder 7)
3	Bitplane (ungerade Nummer) mit Sprite 4 (oder 5)
2	Bitplane (ungerade Nummer) mit Sprite 2 (oder 3)
1	Bitplane (ungerade Nummer) mit Sprite 0 (oder 1)
0	Bitplane (gerade Nummer) mit Bitplane (unger. Nummer)

Die Spritenummern in Klammern bei den Bits 14-9 sind gültig, wenn die entsprechende Spritenummer im CLXCON-Register zur Kollisionserkennung freigegeben ist. Wichtig ist, daß das CLXDAT-Register nach jedem Zugriff automatisch gelöscht wird; falls Sie seinen Inhalt also mehrfach auswerten möchten, sollten Sie diesen zwischenspeichern.

Die Kollisionserkennung ist bei den SimpleSprites wirklich nicht sehr komfortabel, weshalb wir uns nicht näher damit beschäftigen wollen, sondern gleich zur Kollisionsabfrage bei VSprites und Bobs kommen.

Kollisionsabfrage bei VSprites und Bobs

Hier läuft die Kollisionserkennung über die VSprite-Struktur ab, und da VSprite und Bob gemeinsam diese Struktur benutzen, ist auch die Kollisionserkennung gleich. (Wenn wir im folgenden von Gel-Kollisionen sprechen, sind VSprites oder Bobs gemeint.)

Der erste wichtige Eintrag zum Thema Kollisionserkennung ist *gi_collHandler in der GelsInfo-Struktur. Hier tragen Sie einen Zeiger auf eine Tabelle ein, die aus 16 Langworten besteht. Diese Langworte weisen ihrerseits auf Einsprünge in Ihrem Programm, an denen die Routinen zur Kollisionsbehandlung stehen. Sie können also 16 verschiedene Routinen (für 16 Kollisionsereignisse) vorsehen. Die Routinen müssen mit RTS enden, da sie, wie wir gleich sehen, über eine Library-Routine aufgerufen werden.

Damit die Kollisionserkennung funktioniert, müssen auf jeden Fall die beiden Einträge vs_BorderLine und vs_CollMask über die InitMasks-Routine eingerichtet werden. Diese Einträge, die schon im Zusammenhang mit den Bobs besprochen wurden, dienen in erster Linie der Kollisionsabfrage, müssen aber bei Bobs generell belegt werden. Wie die Belegung vor sich geht, entnehmen Sie bitte dem Abschnitt über die Bobs.

Welche Gels welche Kollisionsroutine aufrufen, legen Sie in zwei weiteren Einträgen fest: vs_MeMask und vs_HitMask in der VSprite-Struktur. Jedes Bit dieser Wort-Einträge entspricht einer der 16 Kollisions-Routinen (der mit der selben Bitnummer). Kollidieren zwei Gels, werden die Masken der beiden verglichen, und wenn irgendein Bit in beiden Masken gesetzt ist, wird die entsprechende Routine angesprungen.

Wenn Sie z.B. wünschen, daß Gel X und Gel Y bei Kollision die Routine Nr. 4 aufrufen, setzen Sie bei beiden Gels das Bit 4 in der Me- und HitMask.

Die Unterscheidung der beiden Masken beeinflußt die Reihenfolge der Gels auf dem Bildschirm: Vom Gel, das sich weiter oben auf dem Bildschirm befindet, wird die HitMask zur Erkennung herangezogen, vom weiter unten befindlichen die MeMask. Man kann so verschiedene Routinen anspringen. Diese Funktion wird aber nur recht selten gebraucht; in der Regel stimmen HitMask und MeMask überein.

Eine besondere Rolle spielt die Kollisionsroutine Nr. 0. Sie wird angesprungen, wenn das betreffende Gel, die in der GelsInfo-Struktur festgelegten Randkoordinaten überschreitet.

Bei der Initialisierung der GelsInfo-Struktur brauchen Sie im gi_collHandler-Eintrag lediglich einen Zeiger auf einen 64 Byte großen, leeren Speicherbereich einzutragen. Das Setzen der einzelnen Kollisionsroutinen übernimmt eine Graphics-Routine:

SetCollision	=	**-144 (Graphics-Library)**

***routine**	**a0**	<	Zeiger auf Einsprungadresse der einzutragenden Kollisionsroutine
***gelsInfo**	**a1**	<	Zeiger auf GelsInfo-Struktur, in deren Kollisionstabelle die Routine eingetragen werden soll
type	**d0**	<	Nummer (0-15) der einzutragenden Routine
Erklärung			Trägt die Startadresse einer Kollisionsroutine in die GelsInfo-Struktur eines Rastports ein.

Nun zum Anspringen der Kollisionsroutinen. Es gibt eine Graphics-Routine, die prüft, ob seit dem letzten Aufruf eine Kollision stattgefunden hat. Wenn dem so ist, wird in die betreffenden Routinen aus der CollHandler-Tabelle gesprungen. Es ist also nicht der Fall, daß nach Einrichtung der Tabelle mit SetCollision die Routinen automatisch bei Kollisionen angesprungen werden, sondern nur dann, wenn DoCollision aufgerufen wird. Hat keine Kollision stattgefunden, kehrt DoCollision ohne Aufruf zurück.

DoCollision	=	**-108 (Graphics-Library)**

***rastPort**	**a1**	<	Zeiger auf Rastport, der auf Kollisionen geprüft werden soll
Erklärung			Stellt fest, ob seit dem letzten Aufruf eine Gel-Kollision stattgefunden hat und

springt eventuell in die entsprechende, in der CollHandler-Tabelle eingetragene, Kollisionsroutine.

Denken Sie daran, daß die Kollisionsroutinen mit RTS enden müssen. Vor dem Aufruf der Routine wird der Stack mit einigen Werten belegt. An der Stelle 0(sp), also da, wo der Stackpointer hinzeigt, steht wie gewöhnlich die Rücksprungadresse (damit das RTS funktioniert). Bei 4(sp) steht ein Zeiger auf die VSprite-Struktur des Gels, welches die Kollision ausgelöst hat. Bei 8(sp) steht entweder die Adresse der VSprite-Struktur des Sprite, mit dem der erste zusammengestoßen ist, oder (im Falle der Grenzkoordinaten-Überschreitung) ein Wort, daß folgende Flags enthält:

Hit-Flag	Wert	Bedeutung
TopHit	1	Obere Begrenzung überschritten
BottomHit	2	Untere Begrenzung überschritten
LeftHit	4	Linke Begrenzung überschritten
RightHit	8	Rechte Begrenzung überschritten

Die Werte können auch aufaddiert im Flagwort stehen. Aus den Angaben auf dem Stack kann die Kollisionsroutine feststellen (falls sie es nicht aufgrund der HitMask-Bits schon sowieso weiß), welche Gels womit kollidiert sind.

Zur Demonstration greifen wir das vorige Programm noch einmal auf. Wieder wandern die beiden Bobs über den Bildschirm. Ihre Bewegungsrichtungen werden jetzt aber nicht nur dann umgekehrt, wenn sie den Bildschirmrand erreichen, sondern auch, wenn sie zusammenstoßen.

```
* Programm 7.19 (Auszug): Demo Kollisionserkennung (Beispiel Bobs)

        ...

        lea     collprg,a0      ; Einsprungstelle Kollisionsroutine
        lea     ginfo,a1        ; Zeiger auf GelsInfo
        moveq   #2,d0           ; Kollisionsroutine Nr. 2
        jsr     SetCollision(a6)        ; Kollisionsroutine setzen

        ...

main2:  move.l  gfxbase,a6
        jsr     WaitTOF(a6)

        move.l  a4,a1           ; Abfrage, ob Kollision stattgefunden
        jsr     DoCollision(a6)         ; hat (wenn ja, wird die
                                ; entsprechende Routine automatisch
                                ; angesprungen)
```

```
        ...

collprg:
        neg.w   dir1            ; Kollisionsroutine: Bewegungs-
        neg.w   dir2            ; richtungen umkehren
        rts

        ...
```

Programm 7.19 (Auszug)

7.9 Das IFF-Grafikformat

Die Abkürzung IFF steht für "Interchange File Format" (Austausch-Dateiformat). Das IFF-Format stellt also einheitliche Vorschriften zum Ablegen von Datendateien dar, die somit leicht von verschiedenen Programmen bzw. auch von verschiedenen Computersystemen gelesen werden können.

Eine IFF-Datei ist aus sog. "Chunks" (Datenblöcken) aufgebaut. Im allgemeinen IFF-Format, das für verschiedenste Zwecke benutzt werden kann, gibt es diverse Chunk-Typen mit teilweise recht komplexem Aufbau. Um sie wollen wir uns hier nicht kümmern (sie werden auf dem Amiga z.Zt. auch noch kaum verwendet), wir interessieren uns nur für den Aufbau einer IFF-Grafikdatei.

Neben den IFF-Grafikdateien, genannt ILBM-Dateien (Interleaved Bitmap) gibt es auf dem Amiga noch Typen für Text (FTXT, Formatted Text), Musikstücke (SMUS, Simple Musical Score) und Soundeffekte (8SVX, 8-Bit Sampled Voice).

7.9.1 IFF-Struktur- und Daten-Chunks

Das IFF-Format kennt vier sog. "Struktur-Chunks", die den Aufbau der Datei beschreiben und organisieren. In einer IFF-ILBM-Datei wird nur der einfachste der vier Typen verwendet, der FORM-Chunk. Man kann sagen, ein IFF-Bild besteht nur aus einem Form-Chunk. Jeder Struktur-Chunk beginnt mit folgenden drei Langworten:

Der IFF-Strukturchunk-Header

```
00      dc.l    ckID            ; Kennung des Chunks
04      dc.l    ckSize          ; Länge des Chunks in Bytes
08      dc.l    grpSubID        ; Kennung des Chunk-Typs
12      ds.b    ...             ; Beliebige Datenchunks
```

ckID
Hier steht die Kennung des Chunks als ASCII-Zeichen-Langwort. Beim FORM-Chunk findet sich das Langwort $464F524D.

ckSize
Die Länge des kompletten Chunks in Bytes. Die vier Header-Langworte werden nicht mitgerechnet. Im Prinzip steht hier also die Länge der Datei minus 12. Hinweis: Sollte der Chunk eine ungerade Bytelänge haben, wird ein Füllbyte am Ende angefügt, welches jedoch nicht in der Size-Angabe vermerkt wird. Das gleiche gilt auch für die Längen aller untergeordneten Chunks.

grpSubID
Bei Grafikdateien, mit denen wir uns ausschließlich beschäftigen, steht hier die Kennung ILBM (als Langwort $494C424D).

Nach diesen drei Einleitungs-Langworten, die gewöhnlich zum Test, ob es sich bei einer Datei um eine IFF-Grafikdatei handelt, benutzt werden, folgen die sog. "Daten-Chunks" der Grafik. Es gibt derer 8, die aber nicht unbedingt alle in einer Grafikdatei vorhanden sein müssen. Folgende Chunks, die in der hier aufgeführten Reihenfolge stehen sollten, sind bekannt:

BMHD	Bitmap Header. Enthält grundlegende Informationen über Größe, Position, Farbanzahl usw. der Grafik. Dieser Chunk muß vorhanden sein.
CMAP	Colormap. Enthält die Farbtabelle der Grafik. Dieser Chunk ist nicht unbedingt notwendig, gewöhnlich aber immer vorhanden.
GRAB	Enthält die Koordinaten des "Hot Spots" (sensiblen Punktes) eines Maussprites; findet sich nur in Spezialfällen in einer Grafikdatei.
DEST	Destination. Informiert das lesende Programm über die Zielposition der Grafik und über die Behandlung der einzelnen Planes (PlanePick und PlaneOnOff).
SPRT	Sprite. Enthält die Priorität eines eventuellen Sprites (sehr selten benutzt).
CAMG	Vereinfacht die Einstellung der speziellen Amiga-Grafikmodi HAM, Dual-Playfield und Extra-Halfbright.
CRNG	Color Range. Wird von diversen Malprogrammen (wie DPaint) benutzt, um die eingestellten Color-Cycling-Bereiche mit der Grafik abzuspeichern. Je nach DPaint-Version finden sich 4 (bis DPaint 2) oder 6 (ab DPaint 3) CRNG-Chunks in einer Datei.
BODY	Enthält die eigentlichen Grafikdaten.

Nach diesem Kurzüberblick wollen wir nun die einzelnen Chunks ausführlich besprechen. Jeder Daten-Chunk beginnt, wie der FORM-Struktur-Chunk, mit einem Header, der folgendermaßen aussieht:

Der IFF-Datenchunk-Header

```
00      dc.l    ckID                    ; Kennung des Chunks
04      dc.l    ckSize                  ; Länge in Bytes
08      ds.b    ...                     ; Daten des Chunks
```

ckID
Hier steht wieder das ASCII-Langwort der Chunk-Kennung. Bei BHMD z.B. ist es $424D4844.

ckSize
Die Länge des Datenchunks in Byte. Bei ungerader Bytelänge wird, wie beim FORM-Chunk, ein nicht vermerktes Füllbyte nach dem Chunk eingeschoben.

Auf diese zwei Langworte folgen die eigentlichen Daten des Chunks, die sich von Typ zu Typ unterscheiden. Wir gehen nun auf die einzelnen Chunk-Typen ein.

7.9.2 Die Daten-Chunks im Detail

Der BMHD-Chunk

Nach dem Header findet sich folgender Aufbau:

```
00      dc.w    w                       ; Breite der Grafik
02      dc.w    h                       ; Höhe der Grafik
04      dc.w    x                       ; x-Startkoordinate
06      dc.w    y                       ; y-Startkoordinate
08      dc.b    nPlanes                 ; Anzahl der Bitplanes (Farben)
09      dc.b    masking                 ; Maskierungsverfahren
10      dc.b    compression             ; Komprimierungsverfahren
11      dc.b    pad1                    ; Füll-Byte
12      dc.w    transparentColor        ; Bitmuster der transparenten Farbe
14      dc.b    xAspect                 ; Breite eines Pixels
15      dc.b    yAspect                 ; Höhe eines Pixels
16      dc.w    pageWidth               ; Seitenbreite
18      dc.w    pageHeight              ; Seitenhöhe
20              SIZEOF
```

w, h, x, y
Die Größe (in Pixeln) und Startkoordinate der Grafik. Falls die Größe kein Vielfaches von 16 beträgt, müssen die überzähligen Pixel in ein zusätzlichen Wort geschrieben werden.

nPlanes
Die Anzahl der Bitplanes in der Grafik. Die Farbzahl entspricht 2^{Planes}. Falls hier eine 0 steht, enthält die IFF-Datei keine Grafik. Das ist nützlich, wenn nur die Farbinformation eines Bildschirms gespeichert werden soll.

masking
Hier wird die Kenn-Nummer des verwendeten Maskierungsverfahrens eingetragen. Es gelten folgende Zuordnungen:

Maskierung	Wert	Bedeutung
mskNone	0	Keine Maskierung
mskHasMask	1	Maske in Grafik integriert
mskHasTC	2	Transparente Farbe verwenden
mskLasso	3	Füllen bis zum transparenten Rand

mskNone Es wird kein Maskierungsverfahren verwendet. Die Grafik soll 1:1 in die Zielbitmap kopiert werden.

mskHasMask Die Maske ist in die Grafik integriert und muß verwendet werden. Dieses Maskierungsverfahren kommt nur äußerst selten zur Anwendung.

mskHasTC TC steht als Abkürzung für 'Transparent Color'. Grafikpunkte mit dem Bitmuster, das im Eintrag transparentColor angegeben ist, sollen nicht in der entsprechenden Farbe dargestellt werden, sondern die vorherige Grafik in der Zielbitmap soll an diesen Stellen erhalten bleiben. Dieses Verfahren, daß neben mskNone am häufigsten verwendet wird (Beispiel DPaint), dient zum Einlesen von Grafiken, die keine strenge Rechteckform haben.

mskLasso Die Grafik ist von einem ein Pixel breiten Rand in der transparentColor-Farbe umgeben. Die Grafik soll nach dem Einladen vom Rand aus in der transparenten Farbe ausgefüllt werden, wobei der ein-Pixel-Rand um die eigentliche Grafik als Füllbegrenzung dient. Alle dann eingefärbten Pixel gelten als transparent. Auch dieses Maskierungsverfahren wird nur sehr selten angewandt.

compression
Hier wird das verwendete Komprimierungs-Verfahren verzeichnet. Eine Null (cmpNone) steht für eine unkomprimierte Grafik, die (fast) direkt in die Bitmap eingelesen werden kann. Eine Eins an dieser Stelle besagt, daß die Grafik mit dem einzigen zur Zeit verwendeten Komprimierungsverfahren 'ByteRun' komprimiert wurde (cmpByteRun). Wie dieses Verfahren arbeitet, werden wir gleich sehen.

pad1
Ein Füll-Byte, um die Struktur auf eine gerade Adresse "umzubiegen".

transparentColor
Das Bitmuster der transparenten Farbe (muß nicht vorhanden sein). Es wird nicht direkt ein Farbregister angegeben, da die IFF-Grafik eventuell weniger Bitplanes enthält als die Ziel-Bitmap (siehe planePick und planeOnOff im DEST-Chunk), sondern das Bitmuster in der eigentlichen IFF-Grafik. Sollte die Grafik die gleiche Anzahl Bitplanes haben wie die Zielbitmap, kann man das Bitmuster als Farbregisternummer ansehen.

xAspect, yAspect
Hier wird das Verhältnis Höhe-Breite der Grafik angegeben. Bei "normalen" Grafiken steht im x-Eintrag eine 10 und im y-Eintrag eine 11, ein Pixel hat also in x-Richtung 10/11 der Größe in y-Richtung. Auf dem Monitor ist ein Lores-Pixel also etwas höher als breit. Diese Werte sind nützlich, wenn man die Grafik vergrößern und das Verhältnis beibehalten will.

pageWidth, pageHeight
Diese Einträge geben die Bildschirmauflösung der Grafik, unabhängig von der wirklichen Höhe und Breite, an. Bei einem Lores-PAL-Bild stehen hier also die Werte 320 bzw. 256.

Nun ein Beispiel-BMHD-Chunk einer möglichen Grafikdatei:

```
bmhd:       dc.l    "BMHD"      ; Chunk-Kennung
            dc.l    20          ; Länge: 20 Bytes
            dc.w    320,200     ; Breite und Höhe
            dc.w    0,0         ; Startkoordinaten
            dc.b    5           ; 5 Planes = 32 Farben
            dc.b    2           ; Masierung mskHasTC
            dc.b    1           ; Grafik per ByteRun gepackt
            dc.b    0           ; Füllbyte
            dc.w    0           ; Transparente Farbe: Nummer 0
            dc.b    10,11       ; Verhältnis Breite-Höhe
            dc.w    320,200     ; Bildschirm-Auflösung: Lores NTSC
```

Bild 7.7: Beispiel für einen BMHD-Datenchunk

Noch ein Hinweis: PAL bzw. NTSC beziehen sich auf zwei Monitor-Typen. NTSC-Monitore, wie sie in Amerika üblich sind, können maximal 200 Pixel in vertikaler Richtung darstellen, PAL-Monitore, die wir in Deutschland verwenden, 256 Pixel. Daher bezeichnet man die y-Größe 200 auch als NTSC- und 256 als PAL-Auflösung.

Der CMAP-Chunk

Zunächst wieder der Aufbau:

```
00    dc.b   red0         ; Rotanteil Farbe 0
01    dc.b   green0       ; Grünanteil Farbe 0
02    dc.b   blue0        ; Blauanteil Farbe 0
03    dc.b   red1         ; Rotanteil Farbe 1
04    dc.b   green1       ; Grünanteil Farbe 1
05    dc.b   blue1        ; Blauanteil Farbe 1
..    ....   ......
```

Für jede in der Grafik verwendete Farbe finden sich drei Bytes, je eines für den Rot-, Grün- und Blauanteil der

Farbe. Die Länge der Struktur ist, aufgrund der unbestimmten Farbanzahl, variabel. Die maximale Länge beträgt 32 Farben * 3 Bytes = 96 Bytes.

Eine Schwierigkeit ergibt sich: Aufgrund der Tatsache, daß im Chunk für jede Farbe drei Bytes vorhanden sind, die Farben aber normalerweise in einem Wort (ein Nibble pro ein Farbanteil plus ein unbenutztes Nibble) angegeben werden, muß das lesende Programm die Farbdaten etwas umrechnen. Zudem werden die Intensitäten der einzelnen Anteile auch nicht, wie üblich von 0 bis 15 gezählt, sondern sie reichen in diesem Chunk von 0 bis 255, wobei beim Amiga nur jeder 16. Wert benutzt wird. Man erreicht den Amiga-Intensitätswert also, indem man den Chunk-Wert durch 16 teilt.

Das ganze hat aber einen Sinn: Das IFF-Format soll für verschiedene Computersysteme einheitlich sein, damit die Dateien übertragen werden können. Nun kann es Computersysteme geben, die mehr als 16 Intensitätsstufen pro Farbanteil verarbeiten. Die Aufteilung der Farbanteile auf drei Bytes ist die beste Lösung, um allen Systemen gerecht zu werden.

Nachdem die drei Byte-Werte auf den Bereich 0 bis 15 umgerechnet wurden, müssen sie noch zu den drei Nibbles des Farbwortes "zusammengeklebt" werden. Das geschieht, indem man den Rotwert um zwei Nibbles (oder 8 Bits) nach links schiebt (aus $e würde also $e00), den Grünwert um ein Nibble (vier Bit, aus $a würde $a0) und dann die beiden verschobenen Werte mit dem Blauwert (Beispiel: $5) OR-verknüpft. Das ergibt $e00 OR $a0 OR $5 = $ea5, also genau die Darstellung, die wir auf dem Amiga brauchen. Im Programm sieht das Ganze so aus (d0 bis d2 enthalten die drei Farbanteile aus dem CMAP-Chunk):

```
        asr.b   #4,d0       ; Schieben um 4 = Division durch 16
        asr.b   #4,d1
        asr.b   #4,d2
        asl.w   #8,d0       ; Rotwert zwei Nibbles nach links
        asl.w   #4,d1       ; Grünwert ein Nibble
        clr.w   d3          ; Ergebnisfarbe löschen
        move.w  d0,d3       ; Rotanteil nach d3
        or.w    d1,d3       ; Grünanteil dazuORen
        or.w    d2,d3       ; Blauanteil dazu, fertig
```

Bild 7.8: Umrechnung IFF-Farben in Amiga-Farben

Der GRAB-Chunk

Dieser Chunk, der nur in Spezialfällen in einer IFF-Grafik vorkommt, besteht aus zwei Einträgen:

```
00      dc.w    x               ; x-Koordinate des Hotspots
02      dc.w    y               ; y-Koordinate des Hotspots
04              SIZEOF
```

Das x/y-Koordinatenpaar bezeichnet den Anklickpunkt (Hotspot) einer Maussprite-Grafik relativ zur linken oberen Ecke der Grafik.

Der DEST-Chunk

In diesem Chunk, der wahlweise vorhanden sein kann, sind Informationen über die in der Grafik vorhandenen Bitplanes und darüber, wie die nicht vorhandenen Bitplanes behandelt werden sollen, abgelegt.

```
00      dc.b    depth           ; Anzahl der Grafikplanes
01      dc.b    pad1            ; Füllbyte
02      dc.w    planePick       ; Zu kopierende Planes
04      dc.w    planeOnOff      ; Behandlung der übrigen Planes
06      dc.w    planeMask       ; Verwendete Ziel-Bitplanes
08              SIZEOF
```

depth
Die IFF-Grafik kann weniger Bitplanes umfassen als die Bitmap, in die sie einkopiert werden soll. Hier steht die Anzahl der Planes in der IFF-Grafik.

planePick, planeOnOff
Diese beiden Einträge sind schon aus dem Abschnitt über den Grafikaufbau der Bobs (7.8.3) und aus dem Intuition-Kapitel (Image-Struktur) bekannt. Auch bei IFF-Grafiken gibt es die Möglichkeit, nur bestimmte Planes der Zielgrafik in der IFF-Datei abzulegen. Die Planes, die im planePick-Wort eingeschaltet sind, werden aus der IFF-Grafik in die Zielbitmap kopiert. Die übrigen werden, je nach Zustand des entsprechenden planeOnOff-Bits, komplett mit Eins- bzw. Nullbits gefüllt. (Weitere Informationen siehe vorhin erwähnte Abschnitte.) Sinnvoll ist dies z.B., wenn eine Plane komplett aus 1-Bits besteht. Das planePick-Bit wird gelöscht, das planeOnOff-Bit gesetzt, und schon hat man eine ausgefüllte Bitplane, ohne ein einziges Grafikbyte speichern zu müssen.

planeMask
Hier ist noch einmal angegeben, welche Bitplanes der Zielbitmap vom Kopier- bzw. Füllvorgang betroffen sein sollen.

Noch ein Hinweis: In einer normalen, d.h. DPaint-IFF-Grafik findet sich dieser Chunk nicht.

Der SPRT-Chunk

Dieser Chunk ist sehr einfach. Er wird fast nie benutzt und besteht nur aus einem Eintrag:

```
00    dc.w   SpritePrecedence      ; Priorität des Sprites
02           SIZEOF
```

Falls die IFF-Grafik einen Sprite darstellt, kann in diesem Chunk seine Darstellungspriorität angegeben werden.

Der CAMG-Chunk

Dieser Chunk umfaßt auch nur einen Eintrag, ist aber durchaus nützlich:

```
00    dc.l   ViewModes             ; Verwendeter Grafimodus
04           SIZEOF
```

In diesem Chunk kann der verwendete Grafikmodus eingetragen werden. Die Belegung des ViewModes-Eintrages entspricht den ViewModes in der NewScreen- bzw. ViewPort-Struktur (siehe dort). Der Chunk wird für die Grafikmodi Hires, Extra-Halfbright, Dual-Playfield und HAM eingesetzt, gewöhnlich aber nicht für Interlace. Ob dieser Modus nötig ist, muß das lesende Programm durch Abfrage der Grafikhöhe feststellen (ab 400 Pixel Interlace-Modus). Die Verwendung der ViewModes SPRITES, VP_HIDE und GENLOCK_VIDEO in diesem Chunk ist nicht erlaubt.

Der CRNG-Chunk

Dieser Chunk wird nur von DPaint benutzt. Das Programm speichert darin die eingestellten Color-Cycling-Bereiche direkt mit der Grafik ab. Sein Aufbau ist folgender:

```
00    dc.w   pad1                  ; Reserviert für Erweiterungen
02    dc.w   rate                  ; Cycle-Rate
04    dc.w   flags                 ; Cycle-Flags
06    dc.b   low                   ; Erste Cycle-Farbe
07    dc.b   high                  ; Letzte Cycle-Farbe
08           SIZEOF
```

pad1
Dieser Eintrag ist für zukünftige Erweiterungen reserviert. Derzeit steht hier nur eine 0.

rate
Der Wert, der hier eingetragen werden muß, berechnet sich aus der Anzahl der Farbrotationen pro Sekunde nach folgender Formel: **Rate = RPS / 60 * 16384** (RPS für Rotations Per Se-

cond). Für 30 Rotationen pro Sekunde ergibt sich also: Rate = 30 / 60 * 16384 = 8192.

flags
Folgende Flags sind möglich:

Cycle-Flag	Wert	Bedeutung
RNG_ACTIVE	1	Cycle war beim Speichern eingeschaltet
RNG_REVERSE	2	Farben werden rückwärts rotiert

low, high
Geben die erste bzw. letzte Farbe aus der Palette an, die rotiert werden sollen.

Der BODY-Chunk

Nun haben wir die eigentlichen Grafikdaten erreicht. Für den BODY-Chunk gibt es keine Struktur, der Chunk beginnt sofort mit den Grafikbytes.

Die Grafik ist in einer IFF-Datei auf eine etwas ungewöhnliche Weise abgelegt. Normalerweise erwartet man ja, daß die Grafik bitplaneweise vorliegt, also zuerst alle Bytes der ersten Plane, dann alle der zweiten usw. (wie bei der Intuition-Image-Struktur). Eine IFF-Grafik aber wird zeilenweise abgelegt, also genauso wie die Daten eines SimpleSprite (Zeile 1 Plane 1, Zeile 1 Plane 2, Zeile 2 Plane 1, Zeile 2 Plane 2 usw.). Das Ladeprogramm kann die Grafikbytes nicht einfach "am Stück" in den Bitmap-Speicher einlesen, sondern muß sie zeilenweise auf die Bitplanes aufteilen.

Dieses Speicherverfahren wird wahrscheinlich aus Gründen der Einheitlichkeit (Stichwort Übertragung auf andere Systeme) angewandt; für den Amiga ist es eigentlich recht unpraktisch.

7.9.3 Packen und Entpacken von IFF-Grafiken

Bei der Besprechung des BMHD-Chunks sind wir auf einen Eintrag namens 'compression' gestoßen, der angibt, ob die Grafik komprimiert vorliegt oder nicht. Wenn im Compression-Byte eine Null steht, ist die Grafik nicht komprimiert und kann direkt (unter Beachtung der zeilenweisen Speicherung) in die Bitmap übernommen werden. Ansonsten muß vor der Kopie in die Bitmap "entpackt" werden.

Ein Komprimierungs-Verfahren basiert immer auf der Zusammenfassung gleicher Bytefolgen zu einem "Kürzel". Ein Beispiel: Anstatt 100 aufeinanderfolgender gleicher Bytes (z.B. 0-Bytes) könnte auch einfach die Anzahl und der Wert abgelegt werden (in unserem Beispiel 100 und 0). So werden 100 Bytes zu 2 Bytes komprimiert.

Die Effizienz einer solchen Komprimierung hängt natürlich von der Anzahl gleicher Bytes ab. Bei Grafikdateien ist die Effizienz meist recht hoch, da gewöhnlich eine Menge gleicher Bytes (z.B. Nullen) nacheinander auftreten.

Das IFF-Packverfahren ByteRun arbeitet folgendermaßen: Am Anfang jeder Byte-Gruppe steht ein Byte, das angibt, wieviel gepackte bzw. ungepackte Bytes folgen. Bei **n** ungepackten Bytes wird dabei der Wert **n-1** in das Byte geschrieben. Bei **n** gepackten Bytes ist es der Wert **1-n**, also eine negative Zahl (die im Zweierkomplement-Format abgelegt wird). Die maximale Anzahl Bytes in einer Gruppe liegt, da ein Bit schon belegt ist, bei 128.

Wichtig ist, daß das Zeilenende beim Packen nie überschritten wird, sondern die Bytes in diesem Fall auf zwei Blöcke aufgeteilt werden. Bei gepackten Bytes wird nur die Anzahl und der Wert der Bytes abgelegt. Daher ist das Packen erst ab einer Folge von mindestens zwei gleichen Bytes sinnvoll. Nun die exakte Vorgehensweise beim Packen einer IFF-Grafik:

1. Ausgehend vom aktuellen Byte wird die Anzahl aufeinanderfolgender gleicher Bytes ermittelt. Liegt sie bei zwei oder höher, wird bei Punkt 3 fortgefahren. Ansonsten wird die Länge der Folge ungleicher Bytes ermittelt (suche, bis wieder gleiche Bytes kommen) und bei Punkt 2 fortgefahren.
2. Die **n** ungleichen Bytes werden folgendermaßen abgelegt: Ins erste Byte kommt die Anzahl als **n-1**, ab dem zweiten folgen die ungepackten Bytes. Fortsetzung bei Punkt 4.
3. Die **n** gleichen Bytes werden folgendermaßen abgelegt: Im ersten Byte wird die Anzahl als **1-n** verzeichnet, im zweiten der Wert der gepackten Bytes.
4. Falls das Grafikende noch nicht erreicht wurde, zurück zu Punkt 1.

Wie schon erwähnt, müssen die Bytes auf mehrer Gruppen aufgeteilt werden, sobald das Zeilenende überschritten wird bzw. bei Folgen, die länger als 128 Bytes sind.

Nun die Vorgehensweise beim Entpacken:

1. Ein Byte wird gelesen und sein Vorzeichenbit wird geprüft. Ist das Byte (**n**) positiv, geht es zu Punkt 2, ansonsten zu 3.
2. Es folgen **n+1** ungepackte Bytes, die direkt übernommen werden können. Weiter bei 4.
3. Es folgen **-n+1** gepackte Bytes, deren Wert aus dem nächsten Byte ersichtlich ist (bei der Negation Zweierkomplement beachten!).
4. Wenn das Grafikende noch nicht erreicht ist, zurück zu Punkt 1.

Wichtig: Im Längenbyte der gepackten Grafik hat ein Wert von -128 (= $80) keine Bedeutung und muß überlesen werden.

Als Beispiel wollen wir und nun eine (Hex-)Bytefolge ungepackt und gepackt ansehen:

Ungepackte, originale Bytefolge:

00 00 00 01 02 03 04 00 00 00 02 02 02 01 01 00 00 00 04 03 02 00 00 00

Daraus entsteht die gepackte Folge:

```
FE 00 03 01 02 03 04 FE 00 FE 02 FF 01 FE 00 02 04 03 02 FE 00
 |  |  |  |  |  |  |
 |  |  |__|__|__|__|___ n+1 (=4) ungepackte Bytes
 |  |
 |__|__________________ -n+1 (=3) gepackte Null-Bytes (FE steht für -2)
```

Das sind bei dieser kurzen Folge immerhin schon drei Bytes Ersparnis. Meistens sind die Folgen gleicher Bytes (vor allem Nullen) in einer Grafik noch viel länger. Durchschnittlich kann man mit dem ByteRun-Verfahren eine Grafik um 30-40% komprimieren.

Die programmtechnische Realisierung des Packverfahrens können Sie dem Schluß-Demoprogramm dieses Abschnitts entnehmen.

7.9.4 Beispiel einer kompletten IFF-Datei

Bis jetzt haben wir die einzelnen Chunks getrennt betrachtet. In einer IFF-Datei stehen sie aber alle fein säuberlich hintereinander. Als Beispiel wollen wir uns nun die Struktur einer kompletten IFF-Grafikdatei, wie sie von DPaint erstellt wurde, ansehen (wobei wir die eigentlichen Grafikdaten natürlich weglassen, sonst wären die nächsten 20 Seiten mit Grafikbytes voll).

Die verwendete Grafik hat eine Auflösung von 640x200 Punkten, liegt also im Hires-NTSC-Format vor. Es werden 3 Bitplanes, entspricht 8 Farben, verwendet. Außerdem sind, da die Grafik mit DPaint 2 gespeichert wurde, 4 Color-Cycling-Bereiche eingestellt.

```
iffgraphic:  dc.l   "FORM"       ; Kennung Struktur-Chunk
             dc.l   15488        ; Gesamtlänge des Chunks
             dc.l   "ILBM"       ; Chunk-Typ

             dc.l   "BMHD"       ; Daten-Chunk Bitmap-Header
             dc.l   20           ; Länge 20 Bytes
             dc.w   640,200      ; Größe
             dc.w   0,0          ; Startposition
             dc.b   3,2,1        ; Planes, Masking, Compression
             dc.b   0            ; Füllbyte
             dc.w   0            ; Transparente Farbe
             dc.b   10,11        ; Größenverhältnisse
             dc.w   640,200      ; Bildschirmauflösung
```

```
dc.l    "CMAP"          ; Daten-Chunk Colormap
dc.l    24              ; Länge 24 Bytes
dc.b    0,0,0           ; Farbe 0 (Umrechnung s.o.)
dc.b    240,240,240     ; Farbe 1
dc.b    48,48,32        ; ...
dc.b    0,48,64         ;
dc.b    0,64,80         ;
dc.b    32,80,80        ;
dc.b    16,80,80        ;
dc.b    16,80,64        ; Farbe 7

dc.l    "CAMG"          ; Daten-Chunk CAMG
dc.l    4               ; Länge 4 Bytes
dc.l    4               ; Viewmode Hires

dc.l    "CRNG"          ; Daten-Chunk CRNG
dc.l    8               ; Länge 8 Bytes
dc.w    0               ; Reserviert
dc.w    24              ; Geschwindigkeit
dc.w    1               ; Flag Active
dc.b    23,31           ; Erste und letzte Farbe

dc.l    "CRNG"          ; Daten-Chunk CRNG
dc.l    8               ; Länge 8 Bytes
dc.w    0               ; Reserviert
dc.w    2560            ; Geschwindigkeit
dc.w    1               ; Flag Active
dc.b    23,31           ; Erste und letzte Farbe

dc.l    "CRNG"          ; Daten-Chunk CRNG
dc.l    8               ; Länge 8 Bytes
dc.w    0               ; Reserviert
dc.w    2560            ; Geschwindigkeit
dc.w    1               ; Flag Active
dc.b    0,0             ; Erste und letzte Farbe

dc.l    "CRNG"          ; Daten-Chunk CRNG
dc.l    8               ; Länge 8 Bytes
dc.w    0               ; Reserviert
dc.w    2560            ; Geschwindigkeit
dc.w    1               ; Flag Active
dc.b    23,31           ; Erste und letzte Farbe

dc.l    "BODY"          ; Daten-Chunk Body
dc.l    15167           ; Länge
dc.b    ...             ; Grafikbytes ...
```

Bild 7.9: Beispiel für eine komplette IFF-Datei

7.9.5 Einladen und Anzeigen einer IFF-Grafik

Erschrecken Sie bitte nicht, wenn Sie das nächste Demoprogramm sehen: es ist ca. 7 1/2 Seiten lang. Wir denken aber, daß das Thema IFF-Grafiken so interessant und auch wichtig ist, daß es sich lohnt, das Programm komplett abzudrucken.

Das Programm erwartet als Parameter in der Kommandozeile den Namen der IFF-Grafikdatei. Diese wird geöffnet und eingelesen. Falls dabei nichts schief gelaufen ist, wird die Grafik angezeigt. Durch Druck der linken Maustaste kann sie wieder entfernt werden.

Bevor wir zum Programm kommen, muß noch kurz eine Library-Routine vorgestellt werden, die wir uns quasi bei der Exec-Library "ausleihen": Die CopyMemQuick-Routine zum Kopieren von Speicherbereichen.

CopyMemQuick	=	**-630 (Exec-Library)**

Source	**a0**	<	Start des Quell-Adreßbereichs (muß durch 4 teilbar sein).
Dest	**a1**	<	Start des Ziel-Adreßbereichs (muß durch 4 teilbar sein).
Size	**d0**	<	Länge des Bereichs, der kopiert werden soll (in Bytes, muß durch 4 teilbar sein).

Erklärung

CopyMemQuick ist eine verbesserte Version des CopyMem-Befehls und dient, wie man aus dem Namen unschwer erkennen kann, zum Kopieren von Speicherbereichen. Wenn sich der Quell- und Zielbereich überschneiden, kann nur nach unten kopiert werden (a1 < a0).

```
* Programm 7.20: Anzeigen einer IFF-Grafik

ExecBase      =      4
OpenLib       =      -552
CloseLib      =      -414
Open          =      -30
Close         =      -36
Read          =      -42
Write         =      -48
Seek          =      -66
Output        =      -60
OpenScreen    =      -198
CloseScreen   =      -66
LoadRGB4      =      -192
```

```
WaitTOF       =       -270
AllocMem      =       -198
FreeMem       =       -210
CopyMemQuick  =       -630

        move.l   ExecBase,a6

        movem.l  a0/d0,-(sp)     ; Kommandozeile sichern

        lea      dosname,a1      ; Libs öffnen
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,dosbase

        lea      intname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,intbase

        lea      gfxname,a1
        clr.l    d0
        jsr      OpenLib(a6)
        move.l   d0,gfxbase

        move.l   dosbase,a6

        jsr      Output(a6)      ; Standard-Output ermitteln
        move.l   d0,d4

        movem.l  (sp)+,a0/d0     ; Kommandozeile zurück

        move.b   #0,-1(a0,d0)    ; Schluß-Return durch 0 ersetzen
        move.l   a0,d1
        move.l   #1005,d2        ; Öffnungs-Modus
        jsr      Open(a6)        ; Datei öffnen
        tst.l    d0              ; Fehler?
        bne      chk1            ; Wenn nein

        lea      txt1,a0         ; Text 'Datei nicht gefunden'
        bsr      print           ; ausgeben
        bra      ende            ; Zum Ende

chk1:   move.l   d0,d5           ; Filehandle sichern

        bsr      readlong        ; Erstes Langwort lesen
        cmp.l    #"FORM",buff    ; Ist es "FORM"?
        beq      chk2            ; Wenn ja

chk3:   move.l   d5,d1           ; Datei schließen
        jsr      Close(a6)
        lea      txt2,a0         ; Text 'Keine IFF-Grafikdatei'
        bsr      print           ; ausgeben
        bra      ende

chk2:   moveq    #4,d2           ; Ein Langwort (Chunklänge)
        bsr      skip            ; überspringen
```

```
        bsr     readlong        ; Nächstes Langwort lesen
        cmp.l   #"ILBM",buff    ; Ist es "ILBM"?
        bne     chk3            ; Wenn nein

* Hauptschleife: Daten-Chunks lesen

main1:  bsr     readlong        ; Langwort Chunk-Kennung lesen
        tst.l   d0              ; Anzahl gelesener Bytes 0?
        beq     main2           ; Wenn ja, Dateiende
        cmp.l   #"BMHD",buff    ; BMHD-Chunk?
        beq     bmhd
        cmp.l   #"CMAP",buff    ; Oder CMAP?
        beq     cmap
        cmp.l   #"CAMG",buff    ; Vielleicht ist es "CAMG"?
        beq     camg
        cmp.l   #"BODY",buff    ; Dann aber "BODY"!?
        beq     body

* Unbekannten Chunk überspringen

        bsr     readlong        ; Längen-Langwort einlesen
        move.l  buff,d2         ; Länge nach d2
        bsr     skip            ; 'Länge' Bytes überspringen
        bra     main1           ; Zur Hauptschleife

main2:  move.l  d5,d1           ; Datei schließen
        jsr     Close(a6)

* Grafik darstellen

        cmp.b   #3,chunks       ; Wurden die drei Hauptchunks
        bge     main3           ; gelesen?

        lea     txt3,a0         ; Wenn nein, Text ausgeben
        bsr     print
        bra     ende

main3:  tst.b   compr           ; Ist die Grafik gepackt?
        beq     main5           ; Wenn nein

        bsr     decrunch        ; Entpack-Routine anspringen

main5:  move.l  intbase,a6

        lea     screen,a0       ; Screen öffnen
        jsr     OpenScreen(a6)
        move.l  d0,scr

        move.l  gfxbase,a6

        move.l  scr,a0          ; Zeiger auf Screen nach a0
        add.l   #44,a0          ; Bei Screen+44 liegt der Viewport
        lea     cols,a1         ; Zeiger auf Farbtabelle
        clr.l   d0
        clr.l   d1
        moveq   #1,d0           ; Eins-Bit nach d0
```

```
        move.b  splanes,d1      ; Anzahl Planes nach d1
        asl.l   d1,d0           ; Entspricht 'd0=2^Planes'
        jsr     LoadRGB4(a6)    ; Farben einstellen

        bsr     convert         ; Konviertierungs-Routine

main4:  jsr     WaitTOF(a6)     ; Warte auf Strahlrücklauf
        btst    #6,$bfe001      ; Linke Maustaste gedrückt?
        bne     main4           ; Wenn nein

        move.l  intbase,a6

        move.l  scr,a0          ; Screen schließen
        jsr     CloseScreen(a6)

        move.l  ExecBase,a6

        move.l  grafmem,a1      ; Speicher freigeben
        move.l  grafsize,d0
        jsr     FreeMem(a6)

        tst.b   compr
        beq     ende

        move.l  grafmem2,a1
        move.l  grafsize2,d0
        jsr     FreeMem(a6)

ende:   move.l  ExecBase,a6

        move.l  dosbase,a1      ; Libs schließen und Ende
        jsr     CloseLib(a6)
        move.l  intbase,a1
        jsr     CloseLib(a6)
        move.l  gfxbase,a1
        jsr     CloseLib(a6)

        rts

bmhd:   add.b   #1,chunks       ; Erster Hauptchunk gelesen
        moveq   #4,d2           ; Längen-Langwort überspringen
        bsr     skip
        moveq   #20,d3          ; BMHD-Chunk (Länge 20 Bytes)
        bsr     readbyte        ; einlesen
        move.w  buff,swidth     ; Daten aus Chunk in NewScreen-
        move.w  buff+2,sheight  ; Struktur übertragen
        move.b  buff+8,splanes
        move.b  buff+10,compr
        clr.l   d0              ; Diverse Daten berechnen:
        move.w  swidth,d0       ; Pixelbreite
        asr.l   #3,d0           ; geteilt durch 8
        move.w  d0,lsize        ; ergibt Breite in Bytes
        clr.l   d1
        move.w  sheight,d1      ; Bytebreite mal Höhe
        mulu    d1,d0
        move.w  d0,psize        ; Ergibt Planelänge (Bytes)
```

```
        clr.l    d1
        move.b   splanes,d1      ; Planelänge mal Planeanzahl
        mulu     d1,d0
        move.l   d0,grafsize2    ; Ergibt Gesamtlänge

        clr.w    d0              ; View-Modus
        cmp.w    #640,buff+16    ; Grafik 640 Pixel breit?
        blt      bmhd1           ; Wenn nein
        or.w     #$8000,d0       ; Modus 'Hires' einschalten
bmhd1:  cmp.w    #512,buff+18    ; Grafik 512 Pixel hoch?
        blt      bmhd2           ; Wenn nein
        or.w     #4,d0           ; 'Interlace' dazuschalten
bmhd2:  move.w   d0,sview        ; Viewmodus eintragen
        bra      main1

cmap:   add.b    #1,chunks       ; Zweiter Hauptchunk gelesen
        bsr      readlong        ; Längen-Langwort einlesen
        move.l   buff,d3         ; und nach d3 schreiben
        bsr      readbyte        ; Chunk einlesen
        clr.l    d6              ; Farbanzahl nach d6
        move.b   splanes,d0      ; (per 'd6=2^Planes)
        move.b   #1,d6
        asl.w    d0,d6
        subq     #1,d6           ; Minus 1 wegen dbra

        lea      buff,a0
        lea      cols,a1

* Berechnung der Farbtabelle

cmap1:  clr.w    d0
        clr.w    d1
        clr.w    d2
        move.b   (a0)+,d0        ; Rotwert
        move.b   (a0)+,d1        ; Grünwert
        move.b   (a0)+,d2        ; Blauwert
        asr.w    #4,d0           ; Werte durch 16 teilen
        asr.w    #4,d1
        asr.w    #4,d2
        asl.w    #8,d0           ; Rotwert zwei Nibbles nach links
        asl.w    #4,d1           ; Grünwert ein Nibble nach links
        move.w   d0,d3           ; Werte OR-verknüpfen
        or.w     d1,d3
        or.w     d2,d3
        move.w   d3,(a1)+        ; und in Farbtabelle eintragen

        dbra     d6,cmap1        ; Schleife
        bra      main1

camg:   moveq    #4,d2           ; Längen-Langwort überlesen
        bsr      skip
        bsr      readlong        ; Viewmode-Langwort einlesen
        move.l   buff,d0         ; und wort-weise mit dem Viewmode-
        or.w     d0,sview        ; Eintrag von NewScreen verknüpfen
        bra      main1
```

```
body:   add.b    #1,chunks       ; Dritter Hauptchunk gelesen
        bsr      readlong        ; Längen-Langwort
        move.l   buff,grafsize   ; Größe der Ladegrafik

        move.l   ExecBase,a6

        move.l   grafsize,d0     ; Speicher für Ladegrafik belegen
        clr.l    d1
        jsr      AllocMem(a6)
        move.l   d0,grafmem

        tst.b    compr           ; Grafik gepackt?
        bne      body1           ; Wenn ja
        move.l   grafmem,grafmem2       ; Kopiergrafik=Ladegrafik
        bra      body2

body1:  move.l   grafsize2,d0    ; Speicher für Kopiergrafik belegen
        clr.l    d1
        jsr      AllocMem(a6)
        move.l   d0,grafmem2

body2:  move.l   dosbase,a6

        move.l   d5,d1           ; Grafik-Chunk einlesen
        move.l   grafmem,d2
        move.l   grafsize,d3
        jsr      Read(a6)
        bra      main1


decrunch:
        move.l   grafmem,a2      ; Zeiger auf Quellgrafik
        move.l   grafmem2,a3     ; Zeiger auf Zielgrafik
        move.l   grafsize2,d3    ; Bytelänge der Zielgrafik

        clr.l    d2
decr1:  clr.l    d0              ; Hauptschleife
        move.b   (a2),d0         ; Ein Byte aus gepackter Grafik
        cmp.l    #$80,d0         ; Ist es gleich $80?
        beq      decr5           ; Wenn ja, überspringen
        blt      decr3           ; Wenn kleiner als $80

        move.w   #256,d1         ; Anzahl gepackter Bytes aus
        sub.w    d0,d1           ; Kennbyte berechnen
        move.b   1(a2),d0        ; Byte, das wiederholt wird

decr2:  move.b   d0,(a3)+        ; Gepacktes Byte in Zielgrafik
        addq.l   #1,d2           ; d2 dient als Zielbytes-Zähler
        dbra     d1,decr2

        add.l    #2,a2           ; Quellzeiger um zwei erhöhen
        bra      decr6

decr3:  add.l    #1,a2           ; Quellzeiger erhöhen
decr4:  move.b   (a2)+,(a3)+     ; Schleife: Ungepackte Bytes
        addq.l   #1,d2           ; in Zielgrafik übernehmen
```

```
        dbra    d0,decr4
        bra     decr6

decr5:  add.l   #1,a2           ; Quellzeiger erhöhen
decr6:  cmp.l   d3,d2           ; Bytelänge der Zielgrafik erreicht?
        blt     decr1           ; Wenn nein

        rts

convert:
        move.l  ExecBase,a6

        move.l  scr,a5          ; Ersten Bitplane-Zeiger aus
        lea     192(a5),a5      ; Screen-Struktur nach a5
        move.l  grafmem2,a4     ; Zeiger auf Kopier-Grafik

        clr.l   d2              ; d2: Derzeitige Zeile
conv2:  clr.l   d3              ; d3: Derzeitige Bitplane

conv1:  move.l  a5,a1           ; Erster Bitplane-Zeiger
        move.l  d3,d1           ; Bitplane-Nummer
        asl.l   #2,d1           ; mal 4
        add.l   d1,a1           ; Auf Zeiger aufaddieren
        move.l  (a1),a1         ; Bitplane-Beginn auslesen
        move.l  d2,d1           ; Zeilen-Nummer
        mulu    lsize,d1        ; mal Zeilenbreite
        add.l   d1,a1           ; Auf Zeiger aufaddieren

        move.l  a4,a0           ; Kopierquelle nach a0
        clr.l   d0
        move.w  lsize,d0        ; Bytezahl: Zeilenbreite
        jsr     CopyMemQuick(a6)        ; Kopieren
        add.w   lsize,a4        ; Grafikzeiger erhöhen

        addq    #1,d3           ; Planenummer plus 1
        cmp.b   splanes,d3      ; Alle Planes kopiert?
        blt     conv1           ; Wenn nein
        addq    #1,d2           ; Zeilennummer plus 1
        cmp.w   sheight,d2      ; Alle Zeilen kopiert?
        blt     conv2           ; Wenn nein

        move.l  gfxbase,a6
        rts

print:  movem.l d1-d3,-(sp)     ; SUB Textausgabe für DOS-Write
        move.l  a0,d2           ; *a0 < Zeiger auf Text (0-terminiert)
        clr.l   d3              ;  d4 < Handle der Ausgabedatei
pr1:    addq    #1,d3
        tst.b   (a0)+
        bne     pr1
        subq    #1,d3

        move.l  d4,d1
        jsr     Write(a6)
        movem.l (sp)+,d1-d3
        rts
```

```
readbyte:
        move.l  d5,d1           ; Bytes aus Datei lesen
        move.l  #buff,d2        ; (Anzahl in d3)
        jsr     Read(a6)
        rts

readlong:
        moveq   #4,d3           ; Ein Langwort aus Datei lesen
        bsr     readbyte
        rts

skip:   move.l  d5,d1           ; Bytes überspringen
        clr.l   d3              ; (Anzahl in d2)
        jsr     Seek(a6)
        rts

dosname:        dc.b    "dos.library",0
                even
intname:        dc.b    "intuition.library",0
                even
gfxname:        dc.b    "graphics.library",0
                even
dosbase:        dc.l    0
intbase:        dc.l    0
gfxbase:        dc.l    0

buff:           ds.b    300
chunks:         dc.b    0
compr:          dc.b    0
lsize:          dc.w    0
psize:          dc.w    0

grafsize:       dc.l    0
grafsize2:      dc.l    0
grafmem:        dc.l    0
grafmem2:       dc.l    0

screen:         dc.w    0,0
swidth:         dc.w    0
sheight:        dc.w    0
                dc.b    0
splanes:        dc.b    0
                dc.b    0,1
sview:          dc.w    0
                dc.w    15
                dc.l    0,0,0,0

scr:            dc.l    0
cols:           ds.b    64

txt1:           dc.b    "Datei nicht gefunden!",10,0
txt2:           dc.b    "Datei ist kein IFF-Bild!",10,0
txt3:           dc.b    "Wichtige IFF-Chunks fehlen!",10,0
```

Programm 7.20: Anzeigen einer IFF-Grafik

Wir denken, daß das Programm aufgrund der vorangegangenen ausführlichen Besprechung des IFF-Formats und der Kommentare im Listing gut verständlich ist.

7.10 Die Basisstruktur der Graphics-Library

Auch die Graphics-Library beinhaltet ein paar interessante Einträge in ihrer Basis-Struktur, die wir uns nun ansehen wollen. Wie in jeder Library sind sie an den positiven Offsets, ausgehend von der Basisadresse zu finden (bei den negativen Offsets befindet sich Sprungtabelle für die Library-Routinen).

Neben den, für uns interessanten, Einträgen gibt es allerdings auch eine Menge Daten, die nur systemintern von Bedeutung sind. Auf sie wollen wir nicht näher eingehen.

Die GraphicsBase-Struktur

```
000   ds.b    gb_LibNode,34         ; Library-Struktur
034   dc.l   *gb_ActiView           ; Zeiger auf aktuellen View
038   dc.l   *gb_copinit            ; Zeiger auf Copper-Startup-Liste
042   dc.l   *gb_cia                ; intern
046   dc.l   *gb_blitter            ; intern
050   dc.l   *gb_LOFlist            ; Zeiger auf LOF-Copperliste
054   dc.l   *gb_SHFlist            ; Zeiger auf SHF-Copperliste
058   dc.l   *gb_blthd              ; intern
062   dc.l   *gb_blttl              ; intern
066   dc.l   *gb_bsblthd            ; intern
070   dc.l   *gb_bsbltt1            ; intern
074   ds.b    gb_vbsrv,22           ; Interrupt-Server für Vert. Blank
096   ds.b    gb_timsrv,22          ; Interrupt-Server für Timer
118   ds.b    gb_bltsrv,22          ; Interrupt-Server für Blitter
140   ds.b    gb_TextFonts,14       ; Listenkopf der Textfont-Liste
154   dc.l   *gb_DefaultFont        ; Zeiger auf Standard-Font
158   dc.w    gb_Modes              ; intern
160   dc.b    gb_VBlank             ; intern
161   dc.b    gb_Debug              ; intern
162   dc.w    gb_BeamSync           ; intern
164   dc.w    gb_system_bplcon0     ; intern
166   dc.b    gb_SpriteReserved     ; intern
167   dc.b    gb_bytereserved       ; Füllbyte
168   dc.w    gb_Flags              ; Library-interne Flags
170   dc.w    gb_BlitLock           ; intern
172   dc.w    gb_BlitNest           ; intern
174   ds.b    gb_BlitWaitQ,14       ; Interner Listenkopf
188   dc.l   *gb_BlitOwner          ; Zeiger auf Blitter-Besitzertask
192   ds.b    gb_TOF_WaitQ,14       ; Interner Listenkopf
206   dc.w    gb_DisplayFlags       ; Darstellungsmodus
208   dc.l   *gb_SimpleSprites      ; Zeiger auf SimpleSprites
212   dc.w    gb_MaxDisplayRow      ; Maximalzahl Bildschirmzeilen
214   dc.w    gb_MaxDisplayColumn   ; Maximalzahl Bildschirmspalten
216   dc.w    gb_NormalDisplayRow   ; Standardwert Bildschirmzeilen
```

```
218     dc.w     gb_NormalDisplayCol  ; Standardwert Bildschirmspalten
220     dc.w     gb_NormalDPMX        ; intern
222     dc.w     gb_NormalDPMY        ; intern
224     dc.l    *gb_LastChanceMemory  ; Zeiger auf "Notfall-Speicher"
228     dc.l    *gb_LCMptr            ; intern
232     dc.w     gb_MicrosPerLine     ; intern
234     ds.b     gb_reserved,8        ; Für zukünftige Erweiterungen
242              gb_SIZEOF
```

Die Bedeutung einiger Einträge, z.B. Interrupt-Server oder Listenköpfe, werden im Exec-Kapitel noch genauer erklärt.

Kapitel 8
Die Exec-Library

Die Listen

Speicherverwaltung

Das Multitasking

Das Message-System

Libraries

Devices

Interrupts

Residents

Spezialfunktionen

Die Basisstruktur der Exec-Library

In diesem Kapitel soll es um die Aufgaben der Exec-Library gehen und die Funktionen, die sie dem Programmierer anbietet. Exec (Amiga's Multitasking **Exec**utive) ist der Kern, auf dem das Betriebssystem aufgebaut ist. Die Komplexität dieser Library ist bedingt durch die hohe Flexibilität der Verwaltung, die bei einem Multitasking-System erforderlich ist. Im Gegensatz zu anderen Betriebssystemen, die auf Grund ihrer statischen Form nur schwer erweitert werden können, sind bei Exec fast keine Grenzen gesetzt.

Alle wichtigen Bereiche werden direkt von Exec beherrscht, wie z.B. die Verwaltung der Tasks, Messages, Devices, Resources, Libraries und anderes. Als Programmierer ist man, wenn man nicht direkt die Hardware programmieren will, auf die Funktionen der Exec-Library angewiesen. Ich erinnere nur an die Funktion OpenLibrary, die wir ständig benutzen. Man kann sagen, daß Exec für Programmierer den Schlüssel zum Amiga darstellt.

Wie wir schon von den anderen Libraries wissen, braucht man die Basisadresse, um auf ihre Funktionen zugreifen zu können. Da Exec die wichtigste Library ist und auch, wie schon erwähnt, die OpenLibrary-Funktion enthält, muß ihre Basisadresse an einer festen Stelle abgelegt werden, damit alle Programme auf sie zugreifen können. Diese feste Stelle ist die Adresse 4.

```
...
move.l   4,a6            ; Basisadresse von Exec
...
```

Hat man die Adresse der ExecBase ausgelesen, kann man die Library wie jede andere benutzen.

8.1 Listen

Ein Multitasking-System, welches beim Amiga implementiert wurde, setzt natürlich eine dynamische Verwaltung der Daten voraus. Um den Anforderungen gerecht zu werden, muß ein genaues Buchhaltungs-System aufgebaut werden. Dies geschieht mit sogenannten Listen.

8.1.1 Die Node-Struktur

Das Grundelement einer Liste (oder auch Kette) sind die Node-Strukturen (Knoten-Strukturen), die die Verbindung zwischen den einzelnen Kettengliedern herstellen. Fast jede Datenstruktur, die Exec benutzt, beginnt mit einer solchen Node-Struktur, damit sie in eine Liste aufgenommen werden kann. Die von Exec benutzten Listen sind sogenannte doppelt verkettete Listen, da sie, neben dem Typ, der Priorität und dem Namen, je einen Zeiger auf den nachfolgenden und den

vorangegangenen Knoten enthalten. Dadurch ist es möglich, die Liste vorwärts und rückwärts zu durchlaufen. Die Node-Struktur hat folgendes Aussehen:

Die Node-Struktur:

```
00     dc.l    *ln_Succ            ; Zeiger auf Nachfolger
04     dc.l    *ln_Pred            ; Zeiger auf Vorgänger
08     dc.b     ln_Type            ; Knotentyp
09     dc.b     ln_Pri             ; Priorität des Eintrags
10     dc.w    *ln_Name            ; Zeiger auf den Namen
12              ln_SIZEOF
```

***ln_Succ**
Der Eintrag **ln_Successor** (Nachfolger) enthält einen Zeiger auf die nächste Knoten-Struktur. Durch diesen und den nächsten Zeiger sind die Node-Strukturen miteinander verbunden.

***ln_Pred**
Da es sich um eine doppelt verkettete Liste handelt, wird auch ein Zeiger auf den Vorgänger (Predecessor) benötigt.

ln_Type
Der Typ der Daten, die mit diesem Knoten verwaltet werden, wird in dem Eintrag ln_Type abgelegt. Dazu steht eine ganze Anzahl verschiedener Kennungen zur Auswahl.

Typ	Wert	Bedeutung
nt_Unknown	00	Unbekannter Knoten
nt_Task	01	Programm-Knoten
nt_Interrupt	02	Interrupt-Knoten
nt_Device	03	Device (Gerätetreiber)-Knoten
nt_MsgPort	04	Message-Port-Knoten
nt_Message	05	Message-Knoten
nt_FreeMsg	06	Free-Message-Knoten
nt_ReplyMsg	07	Reply-Message-Knoten
nt_Resource	08	Resource-Knoten
nt_Library	09	Library-Knoten
nt_Memory	10	Memory-Knoten
nt_SoftInt	11	Soft-Interrupt-Knoten
nt_Font	12	Font-Knoten
nt_Process	13	Process-Knoten
nt_Semaphore	14	Semaphore-Knoten
nt_SignalSem	15	Signal-Semaphore-Knoten

Die Struktur der Daten, dessen Typ im Knoten angegeben werden muß, lernen wir im Laufe dieses Kapitels kennen.

ln_Pri
Der Eintrag **ln_Priority** bestimmt die Priorität des Knotens und somit die Position in der Liste. Im Normalfall wird der Eintrag, der zwischen -128 und +127 liegen kann, mit Null initialisiert. Bei speziellen Listen spielt die Priorität der Knoten eine zentrale Rolle, wie z.B. bei der Task-Liste.

Je höher die Priorität eines Tasks eingestellt worden ist, desto mehr Rechenzeit bekommt er vom Prozessor.

***ln_Name**
Der letzte Eintrag der Node-Struktur enthält einen Zeiger auf eine Zeichenkette, die mit einem Null-Byte abgeschlossen ist. Sie enthält den Namen des Knotens und dient zur Identifikation der Daten.

8.1.2 Die List-Struktur

Mit der Node-Struktur haben wir die Glieder der Kette kennengelernt. Nun fehlt uns nur noch der Anfang bzw. das Ende. Hierzu benutzt man die List-Struktur, die aus der Node-Struktur entstanden ist. Sie repräsentiert den Anfang als auch das Ende einer Liste. Im Unterschied zu der Node-Struktur enthält sie keine Daten und dient lediglich Verwaltungszwecken.

Die List-Struktur:

```
00    dc.l   *lh_Head          ; Zeiger auf ersten Knoten
04    dc.l   *lh_Tail          ; immer 0
08    dc.l   *lh_TailPred      ; Zeiger auf letzten Knoten
12    dc.b    lh_Type          ; Typ der Liste
13    dc.b    lh_Pad           ; Füllbyte
14            lh_SIZEOF
```

***lh_Head**
Der erste Eintrag der List-Struktur ist ein Zeiger auf das erste Glied (den ersten Knoten) der Liste. Wenn die Kette leer ist, also keine Nodes enthält, muß hier ein Zeiger auf den Eintrag *lh_Tail stehen. Damit ist die Liste als leere Liste gekennzeichnet.

***lh_Tail**
Der Wert für *lh_Tail ist auf Null festgelegt, und kennzeichnet den Anfang bzw. das Ende einer Liste.

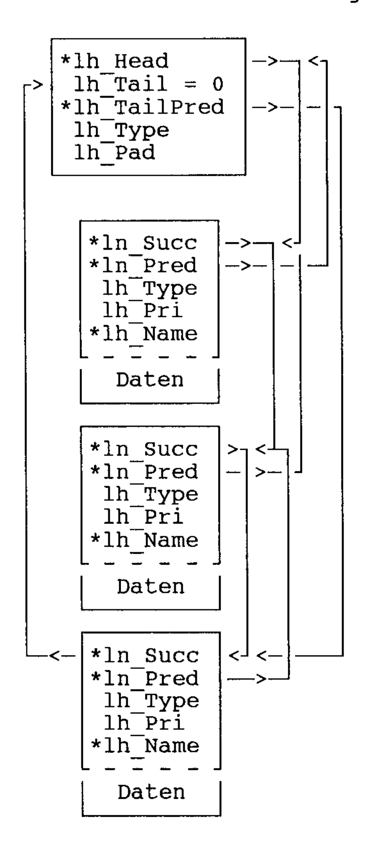
***lh_TailPred**
Da es sich um eine doppelt verkettete Liste handelt, muß natürlich auch im Listenkopf ein Zeiger auf den Vorgänger eingetragen sein. Bei einem Listenkopf ist der Vorgänger der letzte Eintrag der Liste. Steht hier ein Zeiger auf *lh_Head, so handelt es sich um eine leere Liste.

lh_Type
Der vorletzte Eintrag bestimmt den Typ der Daten, die in dieser Liste organisiert werden. Die möglichen Einstellungen sind identisch mit denen des ln_Type-Eintrages der Node-Struktur.

lh_Pad
Der Eintrag lh_Pad hat keinen speziellen Sinn. Er dient lediglich als Füllbyte, da der PC (Programm Counter) nach dem letzten Daten-Byte auf einer ungeraden Adresse steht.

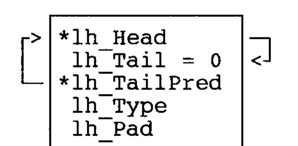
Das Verständnis des Listenprinzips ist unbedingt notwendig, um die nächsten Kapitel zu begreifen. Deshalb sollen die folgenden beiden Skizzen die Anwendung verdeutlichen.

Eine Liste kann beliebig viele Einträge enthalten. Die einzelnen Einträge sind, obwohl sie verstreut im Speicher liegen können, hintereinander wie auf einer Perlenkette aufgereiht. Mit Hilfe der Zeiger *ln_Succ und *ln_Pred kann man sich von Eintrag zu Eintrag hangeln. Dabei bildet die List-Struktur den Anfang und das Ende. Ihr erster Eintrag zeigt auf den ersten Knoten der Liste. Dessen erster Eintrag zeigt auf den nächsten Knoten usw. Um das Ende der Liste zu kennzeichnen, zeigt der *ln_Succ Eintrag der letzten Node-Struktur der Liste auf den zweiten Eintrag des Listenkopfes (lh_Tail=0). Wie *ln_Succ die Adresse des Nachfolgers enthält, so ist *ln_Pred mit der Adresse des Vorgängers belegt. Dabei enthält der erste Ketteneintrag an dieser Stelle einen Zeiger auf den Listenkopf. Versucht man jetzt, den Zeiger auf den nachfolgenden Eintrag zu bekommen, erhält man wiederum einen Null-Wert, der das Ende der Kette kennzeichnet.



Wie man erkennt liegt die Flexibilität einer Kette in den verwendeten Zeigern begründet. Erst sie erlauben es, ohne großen Aufwand einzelne Glieder aus der Kette zu entfernen oder einzufügen. Doch genau in diesem Punkt ist auch der Nachteil einer Kette zu finden. Um einen bestimmten Eintrag zu erreichen, muß man sie sequenziell durchlaufen, da die einzelnen Daten nicht hintereinander im Speicher liegen und ihre Position nicht errechnet werden kann.

Eine leere Kette besteht nur aus der List-Struktur, dessen erster Eintrag (*lh_Head) auf den zweiten *lh_Tail zeigt. Außerdem enthält *lh_TailPred die Adresse von *lh_Head, da es keinen Vorgänger gibt. Um zu testen ob es sich um eine leere Kette handelt, überprüft man entweder, ob der Eintrag *lh_Head auf einen Null-Wert zeigt, oder ob in *lh_TailPred die Adresse des Listenkopfes enthalten ist.



8.1.3 Exec-Routinen zur Listenverwaltung

Um die Verwaltung dieser Listen zu vereinfachen bietet uns Exec ein reichhaltiges Angebot an Funktionen, die zur Manipulation von Ketten geeingnet sind, an.

AddHead	=	**-240 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die Liste, in die der Knoten eingebunden werden soll.
***Node**	**a1**	<	Zeiger auf die Node-Struktur, die in die Liste aufgenommen werden soll.

Erklärung Wie der Name der Funktion schon verrät wird die angegebene Node-Struktur am Anfang der angegebenen List-Struktur eingetragen.

AddTail	=	**-246 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die Liste, in die die Knoten-Struktur aufgenommen werden soll.
***Node**	**a1**	<	Zeiger auf den Knoten, der in die Liste aufgenommen werden soll.

Erklärung Mit der Funktion AddTail kann man eine Node-Struktur an das Ende einer Liste anfügen. Dies geht auch wenn man bei Insert als Predecessor den Eintrag lh_TailPred des Kopfes der Liste angibt.

Enqueue	=	**-270 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die Liste, in die der neue Eintrag aufgenommen werden soll.
***Node**	**a1**	<	Zeiger auf eine Node-Struktur, die in die Liste aufgenommen werden soll.

Erklärung Die Funktion Enqueue trägt die angegebene Node-Struktur in eine Liste ein. Die Position ist dabei abhängig von der Priorität, die der Knoten hat. Der erste Knoten der Kette besitzt die höchste Priorität.

FindName	=	**-276 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die Liste, in der nach dem Knoten mit einem bestimmten Namen gesucht werden soll.
***Name**	**a1**	<	Zeiger auf den Namen, der mit einem Null-Byte enden muß.
***Entry**	**d0**	>	Nachdem die Funktion aufgerufen worden ist, erhält man in d0 einen Zeiger auf die Node-Struktur mit dem gesuchten Namen oder eine Null, wenn kein Eintrag mit dem angegebenen Namen gefunden worden ist.

Erklärung Die Funktion FindName sucht in einer Liste nach einem Knoten mit dem angegebenen Namen.

Insert	=	**-234 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die Liste, in die der Knoten eingefügt werden soll.
***Node**	**a1**	<	Zeiger auf den Knoten, der in die Liste aufgenommen werden soll.
***Predecessor**	**a2**	<	Wird kein Zeiger auf ein Predecessor übergeben (Null), dann wird der Knoten direkt nach dem Listenkopf (Header) eingefügt, andernfalls wird er nach dem hier angegebenen Knoten eingebunden. Der Eintrag in a0 (Liste) ist dann nicht notwendig.

Erklärung Der Knoten wird in die angegebene Liste bzw. nach dem definierten Knoten eingefügt.

RemHead	=	**-258 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die Liste, aus der die erste Node-Struktur entfernt werden soll.
Erklärung			RemHead entfernt den ersten Knoten aus der angegebenen Liste.

Remove	=	**-252 (Exec-Library)**

***Node**	**a1**	<	Zeiger auf eine Node-Struktur, die entfernt werden soll.
Erklärung			Durch Remove kann man einen Knoten aus einer Liste entfernen. Durch das Prinzip der doppelt verketteten Listen muß nur ein Zeiger auf den zu entfernenden Knoten angegeben werden.

RemTail	=	**-258 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die Liste, dessen letzter Eintrag entfernt werden soll.
Erklärung			Parallel zur Funktion AddTail kann man natürlich auch den letzten Knoten einer Liste entnehmen.

Zum Schluß möchte ich nochmal betonen, daß die Listen eine grundlegende Struktur von Exec sind. Das Verständnis und der problemlose Umgang mit ihnen ist sehr wichtig !

8.2 Speicherverwaltung

Die Speicherverwaltung ist einer der wichtigsten Bestandteile eines Multitasking-Betriebssystems. Im Gegensatz zu Monotasking-Systemen, die ihren ganzen Speicher einem Programm zur Verfügung stellen müssen/können, muß die Verwaltung bei Multitasking-Systemen wesentlich flexibler sein. Das Problem liegt darin, daß mehrere Programme gleichzeitig (bzw. schnell hintereinander) bearbeitet werden und alle einen Teil des Speichers benutzen. Wenn ein Task (Programm) Speicher braucht, um Daten abzulegen, wendet er sich an Exec, welches ihm einen Bereich zuweist, der noch frei ist.

Der Arbeitsspeicher den der Amiga zur Verfügung stellt, teilt sich in zwei verschiede Bereiche auf. Diese Bereiche heißen CHIP- und FAST-RAM.

CHIP-RAM Jeder Amiga besitzt in seiner Grundausstattung einen Bereich CHIP-RAM, der je nach Modell verschieden groß ausfällt. Läßt man den A1000 außer Acht, kann man von einer Größe von minimal 512KB ausgehen. Neuere Versionen besitzen bis zu 2MB. Das Besondere an diesem Bereich ist, daß er nicht nur vom Hauptprozessor (MC68000), sondern auch von den Spezialchips, angesprochen werden kann. Diese Chips sind für die Grafik- und Tonausgabe, sowie für die Diskettenzugriffe verantwortlich. Daten für diese Chips müssen in diesem Speicher abgelegt werden.

FAST-RAM Neben dem CHIP-RAM steht uns das FAST-RAM zur Verfügung (In der Grundausstattung des A500 ist dieser Speicherbereich allerdings nicht enthalten). Der Unterschied zum CHIP-RAM besteht darin, daß ausschließlich der Prozessor auf das FAST-RAM zugreifen kann. Das hat den Vorteil, daß dieses RAM schneller ist. Jedoch sollten Grafik-, Ton- oder andere Daten, auf welche die Customchips zugreifen, nicht in diesem Bereich abgelegt werden.

8.2.1 Speicherverwaltung mit der MemHeader-Struktur

Diese beiden Bereiche werden von Exec getrennt in zwei sogenannten Memory Region Header (MRH) verwaltet. Je nachdem, ob das System über FAST-RAM verfügt oder nicht, werden eine oder zwei MRH-Strukturen angelegt. Die erste dieser Strukturen steht direkt im Anschluß an die Exec-Base-Struktur.

Schon jetzt treffen wir wieder auf die Node-Struktur. Sie ist, wie bei den meisten anderen Datenstrukturen auch, in der MH-Struktur enthalten und dient zur Verwaltung.

MemHeader-Struktur:

```
00    dc.l   *mh_Succ          ;┐
04    dc.l   *mh_Pred          ;│
08    dc.b    mh_Type          ;│ Node-Struktur
09    dc.b    mh_Pri           ;│
10    dc.l   *mh_Name          ;┘

14    dc.w    mh_Attributes    ; Typ des Speichers
16    dc.l   *mh_First         ; Zeiger auf freien Block
20    dc.l   *mh_Lower         ; Zeiger auf den Anfang
24    dc.l   *mh_Upper         ; Zeiger auf das Ende
28    dc.l    mh_Free          ; Größe des Speichers
32            mh_SIZEOF
```

***mh_Succ**
Zeiger auf die nachfolgende MH-Struktur.

***mh_Pred**
Zeiger auf die vorangegangene MH-Struktur.

mh_Type
Typ der Daten, die dieser Knoten verwaltet. Hier muß sinngemäß der Wert von nt_Memory (siehe ln_Type) also 10 eingetragen werden.

mh_Pri
Die Priorität wird unterschiedlich zwischen FAST- und CHIP-RAM gewählt. FAST hat einen Prioritätswert von 0 und CHIP von -10.

***mh_Name**
Hier wird der Zeiger auf eine Zeichenkette eingetragen, die den Namen des Knoten enthält.

mh_Attributes
Unter dem Eintrag mh_Attributes versteht man die Angaben, die den Speichertyp bestimmen. Hier stehen zwei verschiedene Typen zur Auswahl.

MEMF_CHIP	02
MEMF_FAST	04

***mh_First**
Der nächste Eintrag ist ein Zeiger auf eine weitere Struktur, die sogenannte Chunk-Struktur. Ihr Aufbau wird im Anschluß erklärt.

***mh_Lower, *mh_Upper**
Die Einträge *mh_Lower und *mh_Upper enthalten den Anfang sowie das Ende des zu verwaltenden Speichers.

mh_Free
Der Eintrag mh_Free enthält die Anzahl der Bytes, die insgesamt noch frei sind.

Nun wollen wir die angesprochene Chunk-Struktur untersuchen. Sie besteht lediglich aus zwei Einträgen und wird direkt in die freien Bereiche des Speichers geschrieben. Auch diese Struktur wird in einer Liste organisiert. Es handelt sich hierbei jedoch um eine einfach verkettete Liste, d.h. die Elemente der Kette sind nur mit einem Zeiger auf den nächsten Eintrag ausgestattet.

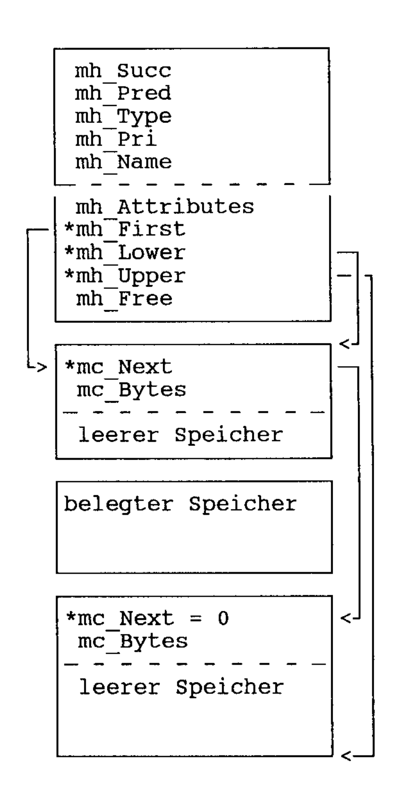
MemChunk-Struktur:

```
00      dc.l    *mc_Next              ; nächste Struktur
04      dc.l     mc_Bytes             ; länge des Chunks
```

***mc_Next**
Der Zeiger *mc_Next zeigt auf die nächste MemChunk-Struktur, die den nächsten freien Bereich verwaltet. Liegt keine weitere Struktur vor, steht hier eine Null.

mc_Bytes
Die Größe des Bereichs, der von der MemChunk-Struktur verwaltet wird, ist in mc_Bytes angegeben.

Die folgende Skizze soll zeigen, wie Exec den Speicher verwaltet.



MemHeader-Struktur

(Node-Struktur um die MemHeader-Strukturen in eine Liste aufzunehmen)

Angabe über den Speichertyp.
Zeiger auf ersten MemChunk.
Unterste Speichergrenze.
Oberste Speichergrenze.
Anzahl der freien Bytes.

MemChunk-Struktur
Zeiger auf nächste Struktur.
Anzahl freier Bytes.

(freier Speicher)

(belegter Speicher)

MemChunk-Struktur
Null => keine weitere Struktur.
Anzahl freier Bytes.

(freier Speicher)

Natürlich kann auch vom Programmierer eine MemHeader-Struktur benutzt werden, um einen eigenen Speicherbereich zu ver-

walten. Unterstützt wird er dabei von zwei Funktionen die von Exec bereitgestellt werden.

Allocate	=	**-186 (Exec-Library)**

***MemHeader**	**a0**	<	Zeiger auf eine MemHeader-Struktur, aus deren verwaltetem Speicher ein Teil belegt werden soll.
ByteSize	**d0**	<	Größe des zu belegenden Speichers.
***MemPtr**	**d0**	>	Nach dem Aufruf der Funktion erhält man einen Zeiger auf den belegten Speicher oder eine Null, falls ein Fehler aufgetreten sein sollte, zurück.
Erklärung			Durch Allocate kann ein Speicherbereich mit angegebener Größe im Bereich der angegebenen MemHeader-Struktur belegt werden.

Deallocate	=	**-192 (Exec-Library)**

***MemHeader**	**a0**	<	Zeiger auf die MemHeader-Struktur.
***MemPtr**	**a1**	<	Zeiger auf den Speicherbereich, der freigegeben werden soll.
ByteSize	**d0**	<	Größe des Speichers, der freigegeben werden soll.
Erklärung			Der durch die Funktion Allocate belegte Speicher kann hiermit wieder freigegeben werden.

8.2.2 Speicherbelegung mit AllocMem und FreeMem

Nachdem wir die Strukturen und die Funktionen kennengelernt haben, mit denen Exec den Speicher verwaltet, wenden wir uns nun den Möglichkeiten zu, die dem Programmierer angeboten werden, um Speicher zu belegen. Sicherlich kann man auch mit den Funktionen Allocate und Deallocate arbeiten, wenn man die Adresse der MemHeader-Struktur kennt. Jedoch gibt es einen bequemeren Weg, mit dem man Speicher einer bestimmten Größe vom System anfordern und wieder freigeben kann. Neben der Größe des Speichers können Sie zusätzlich den Speichertyp definieren. Hierbei stehen fünf Flags zur Auswahl.

MEMF_PUBLIC Durch dieses Flag wird dem System mitgeteilt, daß die Daten dieses Speicherbereichs nicht verschoben werden dürfen. Zur Zeit ist die Funktion, daß Exec eigenhändig Daten ver-

schiebt, nicht implementiert. Deshalb kann man dieses Flag außer Acht lassen.

MEMF_CHIP Durch MEMF_CHIP wird ein Bereich des CHIP-RAMs angefordert. Das ist wichtig, wenn man Grafikdaten oder andere von den Coprozessoren benutzte Daten laden will.

MEMF_FAST Wenn man Speicher aus dem FAST-RAM-Bereich zugewiesen haben will, muß man dies durch das Flag MEMF_FAST angeben. Wird weder MEMF_CHIP noch MEMF_FAST definiert, hat Exec freie Auswahl. Lediglich die Priorität des Speichers gibt an, welcher Bereich benutzt wird. Normalerweise besitzt das FAST-RAM eine Priorität von 0 und das CHIP-RAM von -10. So wird zunächst das Fast- und dann erst das Chip-RAM benutzt.

MEMF_CLEAR Wenn der Speicherbereich, den man angefordert hat, gelöscht (mit Nullen gefüllt) werden soll, kann man das mit MEMF_CLEAR veranlassen. Ansonsten kann es sein, daß der Speicher noch mit Daten belegt ist (z.B. mit einer MemChunk-Struktur, die den freien Bereich verwaltet hat).

MEMF_LARGEST Durch das Flag MEMF_LARGEST wird der Speicherbereich aus dem größten, zusammenhängenden Bereich genommen.

Wie bei allen Flags kann man auch die MEM-Flags kombinieren, indem man die Werte addiert.

Name	Wert	Bedeutung
MEMF_PUBLIC	$00001	Speicher fest (nicht unterstützt)
MEMF_CHIP	$00002	Speicher aus dem CHIP-RAM
MEMF_FAST	$00004	Speicher aus dem FAST-RAM
MEMF_CLEAR	$10000	Speicher soll gelöscht werden
MEMF_LARGEST	$20000	Speicher aus längsten Block belegen

Die Funktion zum Allokieren bzw. zum Deallokieren haben folgende Parameter.

AllocMem	=	**-198 (Exec-Library)**

byteSize	**d0**	<	Größe des Speicherbereichs, der benötigt wird.
Requirements	**d1**	<	Art des Speichers, den das System bereitstellen soll.

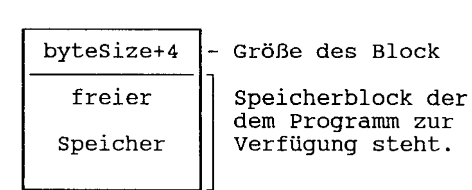
***MemPtr**	**d0**	>	Zeiger auf den Speicherbereich, der von Exec zur Verfügung gestellt wurde oder eine Null, falls kein Speicher der angegebenen Art zur Verfügung steht.
Erklärung			Reserviert einen Speicherbereich mit angegebener Größe vom festgelegten Typ.

FreeMem	=	**-210 (Exec-Library)**

***MemPtr**	**a1**	<	Zeiger auf den Bereich, der wieder freigegeben werde soll.
byteSize	**d0**	<	Größe des Speicherbereichs, der freigegeben werden soll.
Erklärung			Gibt den angegebenen Speicherbereich mit den übergebenen Länge wieder frei.

Man kann durch einen kleinen Trick die Anwendung dieser Funktionen noch komfortabler gestalten. Dazu muß man zwei kleine Routinen schreiben, die das Allokieren und das Deallokieren übernehmen.

Der Trick liegt darin, daß man den angeforderten Bereich um ein Langwort ergänzt, in welches man dann die Länge des gesamten Speicherblocks einträgt. Soll der Speicher nachher wieder freigegeben werden, so benötigt man lediglich die Adresse des Speichers, aus der man die Länge auslesen kann. Durch diesen Trick erspart man sich die Angabe der Länge beim Freigeben des Speichers.



Hier nun die beiden Routinen, die den Speicher belegen und freigeben:

```
*
* ObtainMem
*
* d0 < Größe des zu belegenden Bereichs in Byte
* d1 < Typ des Speicherbereichs
* d0 > Adresse des belegten Speicherbereichs
*

ObtainMem:
        movem.l  d6/a6,-(a7)     ; PUSH
```

```
        addq.l  #4,d0           ; Size += 4
        move.l  d0,d6           ; Größe speichern
        move.l  ExecBase,a6
        jsr     AllocMem(a6)    ; Bereich belegen
        move.l  d0,a0           ; Fehler aufgetreten?
        beq     OMError         ; Ja, dann beenden!
        move.l  d6,(a0)+        ; Sonst Länge in Speicher ablegen
        move.l  a0,d0           ; Adresse zurückgeben
OMError:
        movem.l (a7)+,d6/a6     ; POP
        rts

*
* ReleaseMem
*
* a0 < Zeiger auf Speicherbereich, der freigegeben werden soll
*

ReleaseMem:
        movem.l a6,-(a7)        ; PUSH
        lea     -4(a0),a0       ; Adresse der Länge holen
        move.l  (a0),d0         ; Länge auslesen
        move.l  a0,a1           ; Adresse nach a1
        move.l  ExecBase,a6
        jsr     FreeMem(a6)     ; Speicher freigeben
        movem.l (a7)+,a6        ; POP
        rts
```

Bild 8.1: ObtainMem und ReleaseMem

8.2.3 Speicherreservieren an festen Adressen

Mit den schon beschriebenen Methoden kann man zwar relativ einfach Speicher von Exec bekommen, jedoch ist die Definition einer festen Speicherbelegungs-Adresse nicht möglich. Vielleicht denken Sie jetzt, daß in einem so flexiblen System adressabhängige Speicherbelegung nicht unterstützt wird, doch auch diese Möglichkeit hält uns Exec offen.

AllocAbs	=	**-204 (Exec-Library)**

***Position**	**a1**	<	Zeiger auf die Adresse an der der Speicher belegt werden soll.
byteSize	**d0**	<	Größe des Speichers, der belegt werden soll.
***MemPtr**	**d0**	>	Nachdem die Funktion aufgerufen wurde, erhält man in d0 entweder die Adresse, die in a1 übergeben worden ist, oder eine Null, wenn der gewünschte Bereich nicht zur Verfügung steht.

Erklärung	Durch AllocAbs kann man einen bestimmten Adressbereich belegen. Dies kann jedoch nur funktionieren, wenn der Bereich noch vollständig unbelegt ist.

8.2.4 Zusatzfunktionen der Speicherverwaltung

Besprechen wir nun noch die drei Funktionen, AvailMem, TypeOfMem und AddMemList. Die erste gibt die Größe des angegebenen Speichertyps zurück und die zweite die Attribute des angegebenen Speicherbereichs. AddMemList kann hingegen benutzt werden, um einen neuen, bisher nicht konfigurierten Speicherbereich der MemList (Speicherverwaltungsliste von Exec) hinzuzufügen. Das ist z.B. bei manchen, nicht selbstkonfigurierenden Speichererweiterungen nötig.

AvailMem	=	**-216 (Exec-Library)**

Requirements	d1	<	Typ des Speichers, dessen Größe ermittelt werden soll.
Size	d0	>	Größe des freien Speichers.
Erklärung			Größe des verfügbaren Bereichs des angegebenen Speichertyps ermitteln.

TypeOfMem	=	**-534 (Exec-Library)**

Address	a1	<	Adresse des Speicherbereichs, dessen Attribute ermittelt werden sollen.
Attributes	d0	>	Speichertyp des angegebenen Bereichs, der aus der MemHeader-Struktur ausgelesen wurde.
Erklärung			Durch TypeOfMem wird der Speichertyp des angegebenen Bereichs ermittelt.

AddMemList	=	**-618 (Exec-Library)**

Base	a0	<	Enthält die Basisadresse des neuen Speicherbereichs.
***Name**	a1	<	Zeiger auf eine mit einem Null-Byte abgeschlossene Namensbezeichnung des neuen Speicherbereiches.
Size	d0	<	Größe des Speicherbereichs in Byte
Attributes	d1	<	Attribute des Speicherbereichs
Pri	d2	<	Priorität des Speicherbereichs

Error **d0** > Nach dem Aufruf kann hier eine Fehlernummer oder eine Null als Kennzeichen für fehlerloses Ausführen stehen.

Erklärung Fügt der Speicherverwaltungsliste von Exec (MemList) einen neuen Speicherbereich mit den angegebenen Attributen hinzu.

8.2.5 Speicherverwaltung mit AllocEntry

Noch eine weitere Möglichkeit, Speicher von Exec anzufordern, bietet die AllocEntry-Funktion. Dabei liegt der Vorteil darin, daß man mehrere Blöcke verschiedenen Typs und verschiedener Größen gleichzeitig anfordern kann. Die Bestimmung der Länge und des Typs werden in einer Struktur abgelegt. Bei dieser Struktur handelt es genaugenommen um zwei, die MemList- und die MemEntry-Struktur. Die erste setzt sich wie folgt zusammen:

MemList-Struktur:

```
00      dc.l    *ml_Succ            ;┐
04      dc.l    *ml_Pred            ;│ Node-Struktur für
08      dc.b     ml_Type            ;│ Organisationszwecke
09      dc.b     ml_Pri             ;│
10      dc.l    *ml_Name            ;┘

14      dc.w     ml_NumEntries      ; Anzahl der folgenden
                                    ; EntryStrukturen bzw. die
                                    ; Anzahl der Speicherblöcke

        ; Im Anschluß muß die in ml_NumEntries
        ; festgelegte Anzahl an MemEntry-Strukturen
        ; folgen.
```

***ml_Succ, *ml_Pred, ml_Type, ml_Pri, *ml_Name**
Wie jede Verwaltungs-Struktur von Exec beginnt auch die MemList-Struktur mit einem Knoten. Diese Einträge müssen jedoch nicht vom Programmierer gesetzt werden.

ml_NumEntries
Der Wert ml_NumEntries enthält die Anzahl der Speicherblöcke, die angefordert werden. Für jeden Block muß eine MemEntry-Struktur angelegt werden, die unmittelbar nach der MemList-Struktur folgen muß.

Die MemEntry-Struktur, welche die Größe und den Typ des Speichers angibt, umfaßt zwei Einträge.

MemEntry-Struktur:

```
00      dc.l    meu_Reqs/meu_Addr     ; Typ/Anzahl
04      dc.l    me_Length             ; Größe
```

meu_Reqs/meu_Addr
Das erste Langwort hat zwei Funktionen (deshalb auch zwei Namen). Zunächst wird hier der Typ des Speichers angegeben, der belegt werden soll (**meu_Requirements**). Nachdem die Funktion aufgerufen wurde, erhält man die Adresse einer neuen Struktur, die allerdings den gleichen Aufbau wie die übergebene aufweist. Dieser Eintrag enthält die Adresse des belegten Bereichs (**meu_Address**).

me_Length
Der letzte Wert muß die Länge des Blocks enthalten, der benötigt wird.

Folgende Funktionen stehen für die Anwendung der AllocEntry-Struktur zur Verfügung:

AllocEntry	=	**-222 (Exec-Library)**

***MemList**	**a0**	<	Zeiger auf eine MemList-Struktur.
***List**	**d0**	>	Zeiger auf eine, von AllocEntry neuangelegte, MemList-Struktur aus der man die Adresse des belegten Speicherbereichs auslesen kann.
Erklärung			Durch AllocEntry ist es möglich, eine beliebige Anzahl von verschiedenen Speicherbereichen zu reservieren, die durch die angegebene MemList-Struktur bestimmt sind. Dabei wird eine identische Struktur angelegt in der die Adressen der Bereiche abgelegt werden

FreeEntry	=	**-228 (Exec-Library)**

***List**	**a0**	<	Zeiger auf die von AllocEntry-Funktion angelegte MemList-Struktur.
Erklärung			Durch die Einträge der übergebenen MemList-Struktur wird der belegte Speicher wieder freigegeben.

Bevor wir uns das Demonstrationsprogramm ansehen wollen, möchte ich noch erwähnen, wie die AllocEntry-Funktion abläuft. Dazu folgender Auszug aus dem KickStart-ROM 2.0:

```
AllocEntry:        ; AllocEntry (222)
                   ; a0 < old Entry
                   ; d0 > new Entry
```

```
        MOVEM.L  D2-3/A2-4,-(A7)        ; Register retten
        MOVEA.L  A0,A2            ; a2 =a0

; Zunächst wird die Anzahl der MemEntry-Strukturen gelesen und mit acht
; multipliziert, um auf die Größe der neu anzulegenden Struktur zu
; kommen. Danach wird die Größe um 16 erhöht, da auch die Knoten-
; Struktur nachgebildet werden soll.

        MOVEQ    #0,D3            ; d3 löschen
        MOVE.W   $E(A2),D3        ; Anzahl der MemEntry-Strukturen
        MOVE.L   D3,D0            ; auslesen und nach d0 kopieren
        LSL.L    #3,D0            ; Anzahl * 8 um
        ADDI.L   #$10,D0          ; byteSize += 16
        MOVE.L   #$10000,D1       ; Requirements
        JSR      -$C6(A6)         ; > AllocMem

        MOVEA.L  D0,A3            ; Adresse nach a3
        MOVEA.L  D0,A4            ; a4 = a3
        TST.L    D0               ; Fehler bei Speicherbelegung, dann
        BEQ.S    $F81EE2          ; Routine beenden

; Jetzt wird die neue Struktur mit den "alten" Werten initialisiert und
; der angeforderte Speicher belegt.

        MOVE.W   D3,$E(A3)        ; NumEntries übertragen
        LEA      $10(A2),A2       ; Adresse der neuen MemEntry-Stru.
        LEA      $10(A3),A3       ; Adresse der alten MemEntry-Stru.
        MOVEQ    #0,D2            ; d2 löschen
LF81E8A MOVE.L   0(A2),D1         ; Requirements auslesen
        MOVE.L   4(A2),D0         ; Length auslesen und in neue
        MOVE.L   D0,4(A3)         ; Struktur übertragen
        BEQ.S    LF81EA0          ; Länge = Null, dann weiter
        JSR      -$C6(A6)         ; > AllocMem
        TST.L    D0               ; Fehler bei Belegung des Speichers
        BEQ.S    $F81EB6          ; aufgetreten? Ja, dann verzweigen.

; Nun wird die Adresse des Speichers nur in die neue Struktur
; eingetragen und, falls noch ein Speicherbereich belegt werden soll,
; die Routine erneut abgearbeitet.

LF81EA0 MOVE.L   D0,0(A3)         ; Adresse des Speichers
        ADDQ.L   #8,A2            ; a2 und a3 auf nächste MemEntry-
        ADDQ.L   #8,A3            ; Struktur zeigen lassen.
        ADDQ.W   #1,D2            ; Zähler für die angelegten
        SUBQ.L   #1,D3            ; Strukturen erhöhen und Anzahl
        BNE.S    LF81E8A          ; der gewollten erniedrigen. Noch
                                  ; nicht alle Struk. belegt, dann
                                  ; Routine wiederholen
        MOVE.L   A4,D0            ; Adresse der Struktur übergeben
LF81EB0 MOVEM.L  (A7)+,D2-3/A2-4        ; Register restaurieren
        RTS                       ; Programm beenden

LF81EB6 ; Die Routine, die im Falle eines Fehlers den schon
        ; belegten Speicher freigibt, wollen wir uns an dieser
        ; Stelle schenken.
        ...
```

```
LF81EE2 BSET    #$1F,D0         ; 32. Bit setzen
        BRA     $F81EB0         ; Ende!
```

Bild 8.2: ROM-Auszug der AllocEntry-Routine

Wie dieser Auszug aus dem KickStart-ROM zeigt, wird nicht die angegebene (alte) Struktur mit neuen Werten (meu_Address) gefüllt, wie das in einigen Büchern behauptet wird, sondern es wird eine ganz neue Struktur angelegt. Außerdem erzählt man sich, daß mehrere Strukturen, die durch ihre Knoten verbunden sind, mit einem AllocEntry-Aufruf abgearbeitet werden können. Ich bin mir nicht sicher, ob bei früheren Versionen der AllocEntry-Funktion diese Möglichkeit gegeben war, jedoch ist sie bei der 2.0 Version sicher nicht berücksichtigt (Lediglich beim KickMemPtr ist eine Art Verkettung möglich).

Nun aber das versprochene Demonstrationsprogramm:

```
*
* Kapitel 8
* Demonstrationsprogramm für die Speicherbelegung durch
* AllocEntry und FreeEntry
*

ExecBase        =       4
AllocEntry      =       -222
FreeEntry       =       -228

Start:
        move.l  ExecBase,a6     ; Zeiger auf Exec-Base

        lea     MemStruktur,a0 ; Zeiger auf MemList-Struktur
        jsr     AllocEntry(a6) ; Speicher "bestellen"

        ; Nach der Funktion AllocEntry bekommt man einen
        ; Zeiger auf die neue, von AllocEntry angelegte,
        ; Struktur. Dort sind die Adressen der belegten
        ; Speicherbereiche zu finden. Ansonsten ist die
        ; Struktur identisch mit der übergebenen Struktur.

;       ...

        move.l  d0,a0           ; Adresse der neuen MemList-
        jsr     FreeEntry(a6)   ; Struktur übergeben und
        rts                     ; Speicher freigeben


MemStruktur:
      dc.l   0,0                ;┐
      dc.b   0,0                ;│ Node-Struktur
      dc.l   0                  ;┘
```

```
      dc.w   1                    ; NumEntries

      dc.l   0                    ; Requirements/Addr
      dc.l   300                  ; Length
```

Programm 8.1: AllocEntry und FreeEntry

8.2.6 Speicherbelegung unter Intuition

Zum Abschluß der Speicherfunktionen kommen wir nun zu der AllocRemember- und FreeRemember-Funktion. Zwar sind dies keine Funktionen von Exec, sondern von Intuition, doch sind auch sie für die Speicherverwaltung zuständig und passen gut an diese Stelle.

Auch hier wird eine Struktur benutzt, die sogenannte Remember-Struktur. Sie setzt sich aus folgenden drei Einträgen zusammen:

Remember-Struktur:

```
00    dc.l   *NextRemember        ; Adr d. nächsten Struktur
04    dc.l    RememberSize        ; Größe des Blocks
08    dc.l   *Memory              ; Zeiger auf Block
```

***NextRemember**
Adresse der nächsten Remember-Struktur.

RememberSize
Größe des belegten Speicherblocks.

***Memory**
Adresse des Speichers, den uns Exec zugeteilt hat.

Für das Anlegen und die Initialisierung dieser Struktur ist Exec verantwortlich. Den Programmierer brauchen nur die folgenden beiden Funktionen zu interessieren.

AllocRemember	=	-396 (Exec-Library)

RememberKey	**a0**	<	Unter RememberKey versteht man den Zeiger auf eine Variable (Long), die die Adresse der ersten Remember-Struktur aufnehmen kann. Dabei muß beim ersten Aufruf der Wert dieser Variablen auf Null gesetzt werden. Bei späteren Aufrufen gibt man immer wieder die Adresse dieser Variablen an, wodurch das System die Liste der Remember-Strukturen selbständig ergänzen kann.
Size	**d0**	<	Größe des Speicherbereichs, der belegt werden soll.

Flags d1 < Flags für den Typ des Speichers, der belegt werden soll.

MemBlock d0 > Nachdem die Funktion ausgeführt worden ist, wird die Adresse des Speicherblocks zurückgemeldet.

Erklärung Durch die Funktion AllocRemember wird ein neuer Speicherbereich belegt und in die angegebene Remember-Struktur aufgenommen.

FreeRemember	=	**-408 (Exec-Library)**

RememberKey a0 < Zeiger auf die erste Remember-Struktur der Liste, deren Speicher freigegeben werden soll.

ReallyForget d0 < Boolscher Ausdruck, der angibt, ob der Speicher freigegeben werden soll oder nicht.

Erklärung Durch den Aufruf der Funktion FreeRemember wird der gesamte Speicher, der durch die Remember-Struktur verwaltet wird, freigegeben.

Um den Komfort deutlich zu machen, der uns durch diese beiden Intuition-Funktionen angeboten wird, soll folgendes Programm dienen:

```
*
* Kapitel 8
* Demonstrationsprogramm für AllocRemember und FreeRemember
*

ExecBase      =       4
OpenLib       =       -552
CloseLib      =       -414
AllocRemember =       -396
FreeRemember  =       -408

Start:  move.l   ExecBase,a6     ; Intuition-Library öffnen
        lea      IntName,a1
        jsr      OpenLib(a6)
        move.l   d0,IntBase
        move.l   d0,a6

        lea      RemKey,a0       ; Adresse des RemKey nach a0
        move.l   #512,d0         ; 512KB belegen
        move.l   #0,d1           ; keine MEM-Flags
```

```
        jsr      AllocRemember(a6)      ; Speicher allokieren lassen
        move.l   d0,MemPtr1       ; Zeiger auf Speicher ablegen

        lea      RemKey,a0        ; Adresse des RemKey nach a0
        move.l   #312,d0          ; jetzt 312KB belegen
        move.l   #0,d1            ; wiederum keine MEM-Flags
        jsr      AllocRemember(a6)      ; Speicher allokieren lassen
        move.l   d0,MemPtr2       ; Zeiger auf Speicher ablegen

        ; ... Programm ...

        moveq    #-1,d0
        move.l   RemKey,a0        ; Zeiger auf RemKey dessen Speicher
        jsr      FreeRemember(a6)       ; freigegeben werden soll

        move.l   ExecBase,a6      ; Intuition-Library schließen
        move.l   IntBase,a1
        jsr      CloseLib(a6)
        rts                       ; Fertig !

* Datenbereich

IntName:       dc.b     "intuition.library",0
               even

IntBase:       dc.l     0              ; Intuition-Base-Ptr
RemKey:        dc.l     0              ; Zeiger auf Remember-Struktur
*
*
*       Es ist wichtig, daß beim ersten Aufruf der
*     Funktion Alloc-Remember hier eine Null steht!
*

MemPtr1:       dc.l     0              ; Speicher für die Adresse der
MemPtr2:       dc.l     0              ; belegten Bereiche
```

Programm 8.2: AllocRemember und FreeRemember

8.3 Das Multitasking

Die hervorstechenste Eigenschaft des Amiga-Betriebssystems ist die Möglichkeit, mehrere Aufgaben scheinbar gleichzeitig abzuarbeiten. So kann man z.B. während im Hintergrund ein Raytracing-Programm an einer Grafik rechnet, im Vordergrund mit einem Textverarbeitungsprogramm an einem Brief arbeiten, ohne den anderen Task anzuhalten.

In Wirklichkeit werden die Aufgaben nicht gleichzeitig bearbeitet, da der Amiga in der Regel nur über einen MC68000-Prozessor verfügt. Vielmehr arbeitet die CPU immer nur für einen begrenzten Zeitraum an einem Task und setzt

dann seine Arbeit an einem anderen fort. Durch die hohe Rechengeschwindigkeit sieht es für den Benutzer so aus, als ob die Programme gleichzeitig ablaufen würden. Natürlich entstehen durch dieses Timesharing-System Einbußen bei der Geschwindigkeit der einzelnen Tasks, doch der gebotene Komfort läßt dieses Problem schnell vergessen.

Jedes Programm, das vom Prozessor bearbeitet werden soll, besitzt eine sogenannten TaskControl-Struktur. Durch sie erhält das System die notwendigen Informationen für das Task-Switching (Aufgabenwechseln). Wie alle Daten werden auch die Tasks in Listen organisiert. Exec legt zwei verschiedene Listen für die Tasks an, TaskWait und TaskReady. Dabei verwaltet TaskReady alle Programme, die auf die CPU warten und jederzeit weiterbearbeitet werden können. TaskWait enthält hingegen die "schlafenden" Tasks, die auf eine Meldung warten. Sie werden erst wieder in die Ready-Liste aufgenommen, wenn eine Nachricht empfangen worden ist. Die Header-Strukturen (Listenkopf) dieser Listen stehen im Exec-Datenbereich. Abgesehen davon gibt es noch den Task (ThisTask), der im Augenblick bearbeitet wird. Er gehört zu keiner der beiden Listen.

8.3.1 Die Task-Struktur

Bevor wir näher auf den eigentlichen Switching-Vorgang eingehen, sollten wir uns die Task-Struktur ansehen.

Task-Struktur:

```
00    dc.l   *tc_Succ          ;┐
04    dc.l   *tc_Pred          ;│
08    dc.b    tc_Type          ;│ Node-Struktur
09    dc.b    tc_Pri           ;│
10    dc.l   *tc_Name          ;┘

14    dc.b    tc_Flags         ; Flags für den Task
15    dc.b    tc_State         ; Zustand des Tasks
16    dc.b    tc_IDNestCnt     ; Zähler Interrupt-Disable
17    dc.b    tc_TDNestCnt     ; Zähler Taskswitching-Disab.
18    dc.l    tc_SigAlloc      ; Reaktions-Flags
22    dc.l    tc_SigWait       ; Warte-Flags
26    dc.l    tc_SigRecvd      ; Ankommende Signale
30    dc.l    tc_SigExcept     ; Ausnahmesignale
34    dc.w    tc_TrapAlloc     ; belegte Traps
36    dc.w    tc_TrapAble      ; freie Traps
38    dc.l   *tc_ExceptData    ; Daten für Exception
42    dc.l   *tc_ExceptCode    ; Routine für Exception
46    dc.l   *tc_TrapData      ; Daten für Traps
50    dc.l   *tc_TrapCode      ; Routine für Traps
54    dc.l    tc_SPReg         ; Platz für StackPointer
58    dc.l    tc_SPLower       ; Untere Stack Grenze
62    dc.l    tc_SPUpper       ; Obere Stack Grenze
66    dc.l    tc_Switch        ; Routine für Abgabe
70    dc.l    tc_Launch        ; Routine für Übernahme
```

```
74      ds.b    tc_MemEntry,14        ; Task-MemListe 14 Bytes
88      dc.l   *tc_UserData           ; Zeiger auf eigene Daten
92             tc_SIZEOF
```

***tc_Succ, *tc_Pred, tc_Type, *tc_Name**
Wie fast alle Strukturen fängt auch die Task-Struktur mit einem Knoten an. Er ermöglicht es, den Task in eine der Exec-Listen aufzunehmen.

tc_Flags
Das erste Datenbyte der Struktur enthält Flags, deren Bedeutung in der nachstehenden Tabelle aufgeführt sind.

Taskflag	Wert	Bedeutung
tb_ProcTime	00	(noch nicht benutzt)
tb_StackChk	04	(noch nicht benutzt)
tb_Except	05	Task benutzt Exceptions
tb_Switch	06	tc_Switch-Routine ist installiert
tb_Launch	07	tc_Launch-Routine ist installiert

tc_State
Durch tc_State wird der Status des Tasks angezeigt. Hier sind folgende Werte definiert:

Name	Wert	Bedeutung
ts_Invalid	00	Task ungültig
ts_Added	01	Task ist gerade erstellt worden
ts_Run	02	Task läuft
ts_Ready	03	Task fertig für Ubernahme der CPU
ts_Wait	04	Task wartet auf Signal
ts_Except	05	Task behandelt Exception
ts_Removed	06	Task wird aus dem System entfernt

tc_IDNestCnt, tc_TDNestCnt
Die beiden Einträge Interrupt Disable **Nesting Counter** und **Task** Disable **Nesting Counter** enthalten die Anzahl der Disable- und Forbid-Aufrufe. Wird der Task in den Wartezustand versetzt, benutzt Exec diese Werte, um die Interrupts bzw. den Taskwechsel wieder zuzulassen. Soll der Task später fortgesetzt werden, wird der Ausgangszustand anhand dieser Zähler wieder hergestellt.

tc_SigAlloc
Der Wert tc_SigAlloc enthält einen Langwort-Wert, dessen 32 Bits die belegten Signale enthält. Dabei werden die ersten 16 (0-15) von Exec benutzt!

tc_SigWait
SignalBit, auf den der Task im Augenblick wartet. Natürlich kann man auch mehrere Bits setzen, wodurch die Möglichkeit gegeben ist, auf verschiedene Ereignisse zu warten.

tc_SigRecvd
SignalBits, die empfangen wurden.

tc_SigExcept
Signalbits, die eine Ausnahmezustand auslösen, wenn sie empfangen werden.

tc_TrapAlloc
Belegte Prozessor-Ausnahmen (Traps).

tc_TrapAble
Freigegebene Prozessor-Ausnahmen (Traps).

***tc_ExceptData, *tc_ExceptCode**
Zeiger auf den Datenbereich und die Routine, die bei einer Task-Ausnahmebehandlung benötigt werden.

***tc_TrapData, *tc_TrapCode**
Zeiger auf den Datenbereich und die Routine, die bei einer Prozessor-Ausnahmebehandlung benötigt werden.

tc_SPReg
Wert des StackPointers bei Unterbrechung.

tc_SPLower, tc_SPUpper
Obere und untere Adresse des Task-Stacks.

tc_Switch
Routine, die bei Abgabe des Prozessors ausgeführt wird (Siehe auch tc_Flags).

tc_Launch
Routine, die bei Übernahme des Prozessors ausgeführt wird (Siehe auch tc_Flags).

tc_MemEntry
Unter dem Namen tc_MemEntry sind die Einträge einer List-Struktur zusammengefaßt. Sie wird benutzt, um den vom Task belegten Speicher zu verwalten.

tc_UserData
Der letzte Eintrag kann vom Programmierer beliebig verwendet werden, da er nicht von Exec verwendet wird.

8.3.2 Das Task-Switching

Nachdem wir uns die Task-Struktur angesehen haben, stellt sich die Frage, wie Exec das Umschalten realisiert.

Stellen wir uns vor, der Prozessor bearbeitet gerade einen Task. Nach einiger Zeit (einigen Buszyklen) wird der Ablauf durch einen Interrupt (externe Unterbrechung) gestoppt. Dieser Interrupt hat eine Priorität von 1 bis 6 und ruft die Routine ExitIntr auf. Sie überprüft, ob ein Taskwechsel erlaubt ist.

Danach wird durch die Schedule-Funktion gecheckt, ob ein Task in der TaskReady-Liste vorliegt, der die gleiche oder eine höhere Priorität als der aktive hat, und ob die zugeteilte Zeit des aktiven Tasks abgelaufen ist.

Wenn eine der Anforderungen erfüllt ist, so werden mittels der Switch-Funktion alle wichtigen Informationen wie die Registerinhalte u.Ä. in die Task-Struktur bzw. auf dem Task-Stack abgelegt. Dies geschieht, um das Programm beim nächsten Mal an der gleichen Stelle, mit den selben Werten weiterzuführen. Danach wird, wenn nötig, die tc_Switch Routine angesprungen, um die Abmeldung vom Programm aus vorzunehmen. Der Task wird jetzt wieder in eine der beiden Task-Listen, die von Exec verwaltet werden, aufgenommen.

Um den nächsten Task einzustellen, wird die Funktion Dispatch benutzt. Sie entnimmt den Task mit der höchsten Priorität aus der TaskReady-List und trägt ihn in die ExecBase-Struktur (ThisTask) ein. Weiterhin werden die zwischengespeicherten Werte aus der Task-Struktur ausgelesen, die Register restauriert und die Arbeit an der unterbrochenen Stelle wieder aufgenommen.

Nachdem der Taskwechsel abgeschlossen worden ist, setzt der Prozessor seine Arbeit an der angegebenen Stelle fort, bis er wiederum von einem Interrupt unterbrochen wird.

Hier die benutzten Funktionen:

```
Dispatch    =   -60   ; Task einsetzen
ExitIntr    =   -36   ; Interrupt-Handler
Reschedule  =   -48   ; TSwitch durch SoftInt
Schedule    =   -42   ; Prüfen ob TSwicht nötig
Switch      =   -54   ; Task abbrechen
Wait        =   -318  ; Task in TWait-List
```

Die letzte aufgeführte Funktion (Wait) kann vom Task selbst aufgerufen werden. Sie erzwingt einen Wechsel und fügt den Task in die TaskWait-List ein. Aus dieser Liste wird der Task wieder entnommen, wenn die erwarteten Signale empfangen worden sind (Näheres in den folgenden Kapiteln).

Das Prinzip des Umschaltens dürfte nun etwas klarer geworden sein. Deshalb wollen wir nicht zu weit in die Tiefen des Systems vordringen. Das Wissen reicht vorerst aus, um die Funtkionen, die Exec für die Task-Verwaltung bereitstellt, zu verstehen.

8.3.3 Task-Funktionen

Beginnen wollen wir mit den Funktionen Forbid und Permit. Sie verbieten, daß der laufende Task durch einen anderen abgelöst wird. Das kann sehr wichtig sein, wenn das Programm auf Daten zugreifen will, welche auch von anderen Tasks be-

nutzt werden. Stellt man das Task-Switching nicht ab, so kann es sein, daß während der Task Daten einlesen will, er durch einen anderen abgelöst wird, welcher neue Daten an der gleichen Stelle speichert. Dies kann man durch den Aufruf von Forbid leicht verhindern. Um das System wieder von Mono- auf Multitasking umzuschalten genügt ein Aufruf der Permit-Funktion, und alle anderen Tasks werden wieder berücksichtigt.

Die Forbid-Funktion läßt auch Verschachtelungen zu. So kann man das Task-Switching gleich mehrmals sperren. Dabei wird lediglich der Zähler TDNestCnt (Task Disabel Nesting Counter) erhöht. Entsprechend wird durch Permit der Wert erniedrigt. Enthält der Zähler den Wert -1, so ist das Task-Switching wieder zugelassen.

Forbid	=	**-132 (Exec-Library)**

Erklärung Forbid wird ohne Parameter aufgerufen und verhindert das Wechseln des Tasks durch das Erhöhen des Task Disable Nesting Counter in der Task-Struktur.

Permit	=	**-138 (Exec-Library)**

Erklärung Wie Forbid benötigt auch Permit keine Parameter. Es wird lediglich der Task Disable Nesting Counter erniedrigt. Ist der Zähler kleiner als Null, so ist das Task-Switching wieder erlaubt.

Neben dem Task-Switching kann man auch die Interrupts verbieten. Das Programm wird dann ohne Störungen durch Interrupts abgearbeitet (dadurch kann auch kein Task-Switching mehr vorgenommen werden). Hierzu dient die Funktion Disable. Vergleichbar mit Permit können die Interrupts mittels Enable wieder zugelassen werden. Dabei dient hier der Eintrag IDNestCnt (Interrupt-Disable Nesting Counter) als Zähler.

Disable	=	**-120 (Exec-Library)**

Erklärung Durch die Funktion Disable werden die Interrupts verboten und der Interrupt Disable Nesting Counter erhöht.

Enable	=	**-126 (Exec-Library)**

Erklärung Mit der Funktion Enable wird der Interrupt Disable Nesting Counter erniedrigt.

Sollte der Wert unter Null liegen, werden zusätzlich die Interrupts wieder zugelassen.

Wird der Task durch die Wait-Funktion abgelöst, werden die beiden Werte IDNestCnt und TDNestCnt in die TaskControl-Struktur übernommen.

Die weiteren Funktionen unterstützen den Programmierer dabei, eigene Tasks anzulegen oder zu entfernen bzw. angelegte Task zu bearbeiten.

AddTask = -282 (Exec-Library)

***Task**	a1	<	Zeiger auf die Task-Struktur, die aufgenommen werden soll.
initialPC	a2	<	Hier muß die Adresse angegeben werden, ab der das Programm bearbeitet werden soll.
finalPC	a3	<	Ist der letzte RTS-Befehl ausgeführt worden, so wird zu der Routine verzweigt, deren Adresse hier eingetragen worden ist. Diese Routine kann die "Überbleibsel" des Tasks "aufräumen", wie z.B. belegten Speicher freigeben. Durch einen übergebenen Null-Wert wird zu einer Standard-Routine verzweigt, die den belegten Speicher, der in der MemEntry-Liste eingetragen worden ist, freigibt.

Erklärung Die Funktion AddTask dient dazu, dem System einen neuen Task zu übergeben.

FindTask = -294 (Exec-Library)

***TaskName**	a1	<	Hier kann ein Zeiger auf einen Namen der gesuchten Task-Struktur angegeben werden oder eine Null, wenn die Adresse des eigenen/aktiven Tasks gesucht wird.
Task	d0	>	Nachdem die Funktion aufgerufen wurde, erhält man in d0 einen Zeiger auf die gesuchte Task-Struktur oder eine Null, wenn der gesuchte Task nicht gefunden worden ist.

Erklärung Die Adresse eines durch seinen Namen bestimmten Tasks oder des eigenen Tasks kann mit der Funktion FindTask in Erfahrung gebracht werden. Will man die Adresse des eigenen Tasks (bzw. der ei-

genen Task-Stuktur) wissen, so kann man dazu auch direkt den Wert ThisTask (276) im Exec-Datenbereich auslesen.

RemTask	=	**-288 (Exec-Library)**

***Task**	**a1**	<	Zeiger auf die Task-Struktur, die aus dem System entfernt werden soll.
Erklärung			Um einen Task aus dem System zu löschen kann man sich der RemTask-Funktion bedienen.

SetTaskPri	=	**-300 (Exec-Library)**

***Task**	**a1**	<	Adresse der Task-Struktur, deren Priorität geändert werden soll.
newPriority	**d0**	<	Neue Priorität des Tasks. Der Wert kann zwischen -128 und +127 liegen.
oldPriority	**d0**	>	Nachdem die neue Priorität gesetzt wurde erhält man den Wert der alten in d0.
Erklärung			SetTaskPri setzt die Priorität eines angegebenen Tasks neu.

Um die Anwendung der Funktionen zu demonstrieren, folgt nun ein kleines Beispielprogramm.

```
*
* Kapitel 8
* Demonstrationsprogramm für die Task-Funktionen
*

ExecBase        =       4
FreeMem         =       -210
AllocMem        =       -198
AddTask         =       -282
AllocEntry      =       -222
FindTask        =       -294
AddHead         =       -240

Start:  move.l  ExecBase,a6     ; Stackspeicher belegen
        lea     MemStruktur,a0
        jsr     AllocEntry(a6)
        move.l  d0,a0
        beq     Error
```

```
        move.l   16(a0),d1
        move.l   d1,tc_SPLower   ; Stackpointer eintragen
        add.l    #400,d1
        move.l   d1,tc_SPUpper
        move.l   d1,tc_SPReg

        move.l   #tc_MemEntry,a0        ; Stackspeicher in ME-Struktur
        move.l   d0,a1           ; eintragen
        jsr      AddHead(a6)

        move.l   #TaskStruktur,a1
        move.l   #TaskPrg,a2     ; Zeiger auf Programm
        sub.l    a3,a3
        jsr      AddTask(a6)     ; Task einrichten

        ; ... Programm ...

MainLoop:
        lea      TaskName,a1     ;
        jsr      FindTask(a6)    ; Task suchen
        tst.l    d0              ; wurde Task gefunden?
        bne      MainLoop        ; Ja, dann noch nicht beenden

Error:  rts

TaskPrg:
        move.l   #$40000,d0      ; Schleifenzähler
TPLoop: move.w   d0,$dff180      ; Hintergrundfarbe ändern
        sub.l    #1,d0           ;
        bne      TPLoop          ;
        rts                      ;

* Datenbereich

TaskName:     dc.b    "Mein_TASK",0
              even

TaskStruktur:
              dc.l    0,0           ;┐
              dc.b    1,0           ;│ Node-Struktur
              dc.l    TaskName;┘

              dc.b    0             ; tc_Flags
              dc.b    0             ; tc_State
              dc.b    0             ; tc_IDNestCnt
              dc.b    0             ; tc_TDNestCnt
              dc.l    0             ; tc_SigAlloc
              dc.l    0             ; tc_SigWait
              dc.l    0             ; tc_SigRecvd
              dc.l    0             ; tc_SigExcept
              dc.w    0             ; tc_TrapAlloc
              dc.w    0             ; tc_TrapAble
              dc.l    0             ; tc_ExceptData
              dc.l    0             ; tc_ExceptCode
              dc.l    0             ; tc_TrapData
              dc.l    0             ; tc_TrapCode
```

```
tc_SPReg:      dc.l    0           ; tc_SPReg
tc_SPLower:    dc.l    0           ; tc_SPLower
tc_SPUpper:    dc.l    0           ; tc_SPUpper
               dc.l    0           ; tc_Switch
               dc.l    0           ; tc_Launch
tc_MemEntry:
lh_Head:       dc.l    lh_Tail     ; tc_MemEntry
lh_Tail:       dc.l    0
lh_TailPred:   dc.l    lh_Head
               dc.b    0,0
               dc.l    0           ; tc_UserData

MemStruktur:   dc.l    0,0         ;
               dc.b    0,0         ; Node-Struktur
               dc.l    0           ;

               dc.w    1           ; Num Entries
               dc.l    0           ; Requirements/Addr
               dc.l    400         ; Length
```

Programm 8.3: Demonstration Task-Funktionen

Das Programm startet einen zweiten Task, welcher lediglich die Hintergrundfarbe verändert. Dieser Task ist autark und wird, nachdem er abgearbeitet worden ist, vom System eigenhändig entfernt. Währenddessen kontrolliert der Haupttask, durch die Funktion FindTask, ob der Sub-Task noch existiert.

8.3.4 Verbindung zwischen den Tasks

Die Kommunikation zwischen den Tasks ist sehr wichtig. Wenn z.B. zwei verschiedene Tasks auf den gleichen Speicher zugreifen wollen, so muß dies untereinander abgestimmt werden. Außerdem kann durch Signale ein Task auf spezielle Ereignisse hingewiesen werden. Dies geschieht über die Einträge tc_SigAlloc, tc_SigWait, tc_SigRecvd und tc_SigExcept.

Die unteren Bits des Langwortes tc_SigAlloc (0-15) sind für das System reserviert. Die anderen 16 können frei gewählt werden. Sie sollten jedoch mit dem empfangenden bzw. sendenden Task abgesprochen werden, da ihre Funktionen frei wählbar ist

Zur Verdeutlichung der Kommunikation zwischen Tasks soll die beiden folgenden Programmablaufbeschreibungen dienen:

Task #1	**Task #2**
Vorbereitungen für den Programmablauf tätigen	Vorbereitungen für den Programmablauf tätigen
SignalBit belegen (AllocSignal)	**Adresse von Task #2 ermitteln** (FindTask)
Warten auf Signale (Durch die Funktion Wait wird der Task in die TaskWait-List aufgenommen. Dort kann er nur durch die Funktion Signal wieder entnommen werden.)	**SignalBit an Task #2 senden** (Durch die Funktion Signal kann ein bestimmtes Signal an einen Task gesendet werden. Dabei wird das entsprechende SignalBit im Eintrag tc_SigRecvd gesetzt und der Task aus der TaskWait-List in die TaskReady-List übernommen.)
Nachdem das Signal empfangen worden ist, wird der Task weiter bearbeitet.	alle Vorbereitungen rückgängig machen

Wie man erkennen kann, ist ein Task, der sich durch Wait "schlafen gelegt hat" nur durch einen anderen Task wiederzubeleben. Dabei kann man folgende, von der Exec-Library bereitgestellte, Funktionen benutzen.

AllocSignal	=	**-330 (Exec-Library)**

sigNum **d0** < Nummer eines Signalbits (0-31), welches besetzt werden soll. Um das nächste freie Bit zu benutzen, gibt man -1 an.

sigNum **d0** > Man erhält die Nummer des gesetzten Signalbits oder, wenn alle 32 Bits belegt waren, -1 als Fehlermeldung zurück.

Erklärung Durch die Funktion AllocSignal kann man die Signalbits des laufenden Tasks setzen. (Bezieht sich auf den Eintrag tc_SigAlloc)

FreeSignal	=	**-336 (Exec-Library)**

sigNum **d0** < Nummer des Signalbits, welches wieder freigegeben werde soll.

Erklärung FreeSignal gibt das angegebenen Signalbit des laufenden Tasks frei. (Bezieht sich auf den Eintrag tc_SigAlloc)

SetSignal	=	-306 (Exec-Library)

newSignals	d0	<	Neue Signalbelegung. Die Bits der bisherigen Signalbelegung werden entsprechend dieses Werts verändert.
signalSet	d1	<	Maskenwert für Signale. Nur die Bits, die hier gesetzt sind, werden in der Signalbelegung beachtet.
oldSignals	d0	>	Vorige Signalbelegung
Erklärung			Ändert die Signalbelegung eines Tasks.

Signal	=	-324 (Exec-Library)

*Task	a1	<	Adresse des Tasks, der die angegebenen Signale erhalten soll.
signalSet	d0	<	Signalwert, den der angegebene Task empfangen soll.
Erkärung			Durch die Signal-Funktion läuft die eigentliche Kommunikation ab. Die angegebenen Signalbits werden dem angegebenen Task übergeben. Das heißt, sie werden in den Eintrag tc_SigRecvd eingetragen. Außerdem wird der Task, sofern er auf eine Nachricht wartet, aus der TaskWait- in die TaskReady-List übertragen.

Wait	=	-318 (Exec-Library)

signalSet	d0	<	Maske der Signalbits auf die der Task wartet.
signalSet	d0	>	Hat der Task ein (oder mehrere) Signale erhalten, wird er wieder bearbeitet und die Signalflags werden im Datenregister d0 zurückgeliefert.
Erkärung			Durch die Funktion Wait wird ein Taskwechsel erzwungen und der Task in die TaskWait-Liste von Exec eingetragen. Dort wartet er darauf, daß er von einem anderen Task eins der angegebenen Signale erhält. Erst dann wird er wieder bearbeitet. Liegt eins der Signalbits noch an, so wird die Bearbeitung direkt weitergeführt.

(Alle besprochenen Funktionen sollten nicht während einer Exception aufgerufen werden!)

```
*
* Kapitel 8
* Demonstrationsprogramm für das Signal-System
*

ExecBase        =       4
                        ...
AllocSignal     =       -330
Wait            =       -318
Signal          =       -324

Start:  move.l   ExecBase,a6     ; Adresse der Task-Struktur
        move.l   276(a6),MainTask      ; auslesen

        move.l   #20,d0          ; Signalbit #20 belegen
        jsr      AllocSignal(a6)
        addq     #1,d0           ; d0 += 1 (-1+1= 0)
        beq      Error           ; d0 = 0 => Fehler

        ...      ; Jetzt wird der Task eingerichtet

*                          \|/
        ;          10987654321098765432109876543210
        move.l   #%00000000000100000000000000000000,d0
                                 ; Jetzt warten wir darauf, daß
                                 ; der zweite Task uns das belegte
                                 ; Signalbit sendet.
        jsr      Wait(a6)

Error:  rts                      ; Main-Task beenden

TaskPrg:

        ...      ; Programm des Sub-Tasks

        move.l   ExecBase,a6
        move.l   MainTask,a1     ; Adresse der ersten Task-Struktur
                                 ; nach a1 und Signal abschicken

        ;          10987654321098765432109876543210
        move.l   #%00000000000100000000000000000000,d0
        jsr      Signal(a6)

        rts                      ; Dann wird der Task entfernt

MainTask:           dc.l            0
        ...
```

Programm 8.4: Demonstration Signal-System

Das Demonstrationsprogramm ist lediglich eine abgewandelte Form des Task-Demonstrationsprogramms. Der Unterschied besteht darin, daß der Main-Task erst abgeschlossen wird wenn ein bestimmtes Signal von ihm empfangen worden ist.

8.3.5 Task-Ausnahmen (Exceptions)

Eine Task-Ausnahme ist eigentlich nichts anderes als ein Signal, welches einer besonderen Behandlung unterzogen werden soll. Sollte das empfangene SignalBit mit dem im Eintrag tc_SigExcept abgelegten Wert übereinstimmt, dann wird der Task unterbrochen und die Routine, deren Adresse im Eintrag tc_ExceptCode erwartet wird, ausgeführt. Die Signale, die zu einer Task-Exception führen, können mittels der Funktion SetException gesetzt werden.

SetException	=	**-312 (Exec-Library)**

newSignals d0 < Werte der SignalBits.

signalSet d1 < Maskenwert, der angibt welche Signalbits von der Funktion verändert werden sollen.

```
 10987654 32109876 54321098 76543210
%00001001 00001000 00100000 00001000
```

Durch diese Maske wird z.B. die Veränderung der Signalbits 3, 13, 19, 24 und 27 erlaubt (es wird von 0 bis 31 gezählt). Je nachdem, welchen Wert "newSignals" enthält, werden die Bits gesetzt oder gelöscht.

oldSignals d0 > In d0 erhält man die alten Werte der Signalbits zurück.

Erklärung Durch die Funktion SetException kann man die Signalbits des tc_SigExcept Eintrags des laufenden Tasks ändern. Diese Bits geben an, bei welchen Signalen die Routine tc_ExceptCode ausgeführt werden soll.

Empfängt ein Task ein Signal, welches für eine Task-Ausnahme reserviert ist, werden folgende Schritte unternommen:

Zunächst werden alle wichtigen Register, wie der Programm Counter, das Status Register und die Register d0-7 und a0-6 auf den Stack des Tasks abgelegt. Dann wird die Adresse, die im Eintrag tc_ExceptData eingetragen ist, in das Adreßregister 1 geladen. Über dieses Adreßregister kann dann auf den Datenbereich zugegriffen werden, der für Task-Exceptions an-

gelegt worden ist. In d0 erhält man zusätzlich noch die Signalbits, die den Exceptionzustand ausgelöst haben.

Wurde die Exceptionroutine abgearbeitet, muß sie mittels eines RTS-Befehls verlassen werden. Dann werden die ursprünglichen Registerinhalte wieder hergestellt und der Task fortgesetzt.

Sollte während der Bearbeitung der Exception ein weiteres Signal eine Task-Ausnahme provozieren, so wird sie erst nach der Abarbeitung der ersten berücksichtigt.

8.3.6 Interne Prozessor-Ausnahmen (Traps)

Im Unterschied zu den Task-Ausnahmen werden Prozessor-Ausnahmen nicht durch das Betriebssystem sondern durch den Prozessor selbst geschaffen. So gibt es bei der MC68000-Familie einige Möglichkeiten, das laufende Programm zu unterbrechen und die Arbeit an einer anderen Stelle fortzusetzen, dabei unterscheidet man interne und externe Exceptions.

Als interne Exceptions bezeichnet man Ausnahmezustände, die durch einen Befehl (Trap/TrapV/CHK..) oder einen Fehler (Zero-Divide/Bus-Error..) auftreten können.

Externe Exceptions dagegen werden durch einen externen Baustein verursacht, der an die Interruptleitungen des Prozessors eine bestimmte Signalkombination anlegt (auf diesen Typ gehen wir an dieser Stelle nicht näher ein. Dies geschieht im Abschnitt über die externen Prozessor-Ausnahmen).

Für jede Ausnahme ist ein Vektor vorgesehen, der die Adresse der Behandlungsroutine enthalten sollte. Diese Vektortabelle liegt bei dem MC68000 immer an der Adresse $0-$3FF des Hauptspeichers (bei höheren Prozessortypen kann die Position frei gewählt werden).

Tritt eine Prozessor-Exception auf, so wird der Prozessor in den Supervisor-Modus geschaltet und der aktuelle PC (Programmzähler) sowie das Status-Register auf dem Supervisor-Stack abgelegt. Dann wird die Behandlungsroutine der zugehörigen Ausnahme angesprungen.

In diesem Kapitel sollen zunächst nur die internen Exceptions besprochen werden. Folgende Prozessor-Ausnahmen gibt es:

Bus-Error
Es wurde eine Adresse angesprochen, die keinem Baustein oder Speicher zugeordnet ist.

Address-Error
Es wurde mit einem Langwort oder Wort-Befehl auf eine ungerade Adresse zugegriffen. Achtung: Da beim 68020-Prozessor

ungerade Adressierung erlaubt ist, wird bei ihm auch keine Exception ausgelöst.

```
move.l   #20,$40001      ; Langwort wird an ungerade
                         ; Adresse geladen
```

Illegal-Instruction
Dieser Vektor wird angesprungen, wenn ein illegaler Befehl ausgeführt wird. Dies kann auch mit Hilfe des ILLEGAL-Befehls provoziert werden.

```
illegal                  ; = %0100101011111100 = $4AFC
```

Zero-Divide
Es wurde eine Division durch Null vorgenommen.

```
clr.l    d0              ; d0 mit Null laden
divs     #5,d0           ; und dividieren
```

CHK-Instruction
Die CHK-Instruktion prüft das angegebene Datenregister. Ist der Wert negativ oder kleiner als der angegebene, wird eine CHK-Exception ausgelöst. Die Prüfung des Datenregisters bezieht sich dabei nur auf Wortgröße !

```
move.w   #24,d4          ; Datenregister mit 24 laden
chk      #4,d4           ; Prüfen (CHK -> Exception)
```

TRAPV-Instruction
Durch den TRAPV-Befehl wird das V-Flag (Überlauf) getestet und eine Exception ausgelöst, wenn das Flag auf 0 stand.

```
move.w   #$3FFF,d0       ; Datenregister d0 laden
add.w    #$4001,d0       ; Wert addieren -> Überlauf
trapv                    ; prüft das V-Flag und löst
                         ; eine TRAPV-Exception aus
```

Privilege-Violation
Es wurde im User-Modus ein privilegierter Befehl ausgeführt, der nur im Supervisor-Modus erlaubt ist.

```
ori      #$700,sr        ; Privilegierter Befehl
```

Trace
Diese Exception ist durch das Trace-Bit des StatusRegisters ausgelöst worden. Ist dieses Bit gesetzt, so wird nach jedem Befehl die Trace-Exception ausgeführt (Einzelschritt-Abarbeitung).

Line (%1010) $A-Emulator
Es wurde ein Befehl ausgeführt, der mit der Kodierung %1010 = $A beginnt. Diese Befehlsgruppe hat keine Funktion und löst daher eine Exception aus. Die anderen 12 Bits können z.B. als Parameterübergabe dienen.

```
dc.w    $A405           ; Es reicht auch schon $A000
```

Line (%1111) $F-Emulator
Genau wie beim Line $A-Emulator kann auch durch diese Exception ein Befehl emuliert werden. Jedoch werden die $Fxxx-Befehle beim MC68020-Prozessor für die Programmierung des mathematischen Co-Prozessors benutzt.

```
dc.w    $F27D           ; Es reicht auch schon $F000
```

Trap-Vektor #00-15
Die Exception-Vektoren $80-$BC werden durch den Trap-Befehl des MC68000 ausgelöst. Bei einigen Betriebssytemen dienen die Trap-Befehle zum Verzweigen in Systemroutinen.

```
Trap    #3              ; Ruft den Trap 3 auf
```

Hier die Tabelle der Traps mit ihren Adressen:

Prozessor-Traps:

Vektor	Adresse	Bedeutung
002	$008	Bus-Error
003	$00C	Address-Error
004	$010	Illegal-Instruction
005	$014	Zero-Divide
006	$018	CHK-Instruction
007	$01C	TRAPV-Instruction
008	$020	Privilege-Violation
009	$024	Trace
010	$028	Line %1010 ($A) Emulator
011	$02C	Line %1111 ($F) Emulator
032	$080	Trap-Vektor #00
033	$084	Trap-Vektor #01
034	$088	Trap-Vektor #02
035	$08C	Trap-Vektor #03
036	$090	Trap-Vektor #04
037	$094	Trap-Vektor #05
038	$098	Trap-Vektor #06
039	$09C	Trap-Vektor #07
040	$0A0	Trap-Vektor #08
041	$0A4	Trap-Vektor #09
042	$0A8	Trap-Vektor #10
043	$0AC	Trap-Vektor #11
044	$0B0	Trap-Vektor #12
045	$0B4	Trap-Vektor #13
046	$0B8	Trap-Vektor #14
047	$0BC	Trap-Vektor #15

Tritt nun eine solche interne Task Exception auf, wird der Prozessor in den Supervisor-Modus gesetzt und der Programmzähler (Langwort/zeigt auf nächsten Befehl) sowie das Status-Register (Wort) auf den Supervisor-Stack gerettet. Dann wird die Behandlungsroutine des zugehörigen Vektors ausgeführt. Normalerweise steht hier die Behandlungsroutine von Exec. Sie legt die Trap-Nummer (Langwort) auf den Stack ab, dann wird die Trap-Behandlungsroutine des Programms (tc_TrapCode) angesprungen.

Skizze des Stacks nach einer Ausnahme

Trap-Nummer	0
	+2
Status-Register	+4
Programmzähler	+6
	+8

Ist die eigene Behandlungsroutine abgearbeitet, muß sie über den RTE-Befehl (**R**eturn from **E**xception) verlassen werden. Dabei sollte man beachten, daß zuerst die Trap-Nummer vom Supervisorstack genommen werden <u>muß</u>!

Es sei noch erwähnt, daß es möglich ist, die anderen beiden Werte, die auf dem SSP abgelegt worden sind, zu verändern. Dabei profitiert man davon, daß die Werte, die auf dem SSP liegen, nach der Trap-Routine wieder geladen werden. So kann man z.B. das Trace-Bit im Statusregister setzen, um ein Programm im Einzelschrittmodus zu durchlaufen!

```
*
* Kapitel 8
* Demonstrationsprogramm für den Gebrauch des tc_TrapCode-
* Eintrags in der Task-Struktur.
*

ExecBase      =       4

Start:  move.l   ExecBase,a6      ; Adresse der ExecBase lesen
        move.l   276(a6),a0       ; ThisTask-Eintrag auslesen

        move.l   #0,46(a0)        ; Keine Daten (tc_TrapData=0)
        move.l   #PException,50(a0)    ; Zeiger auf Routine
                                  ; (tc_TrapCode = #PException)

        trap     #1               ; Trap aufrufen

        rts                       ; Programmende


PException:                       ; Trap-Routine
        move.l   d0,-(a7)         ; d0 sichern

        move.l   #$7000,d0        ; Schleifenwert nach d0
Loop:   move.w   d0,$dff180       ; d0 in Color00 schreiben
        sub.l    #1,d0            ; d0 verringern
```

```
        bne     Loop            ; nicht Null, dann verzweigen
                                ; zu Loop
                                ; Bevor man das Programm
                                ; fortsetzt muß der Stack
        add.l   #8,a7           ; bearbeitet werden.
                                ; TrapNummer und der Wert des
                                ; Datenregisters 0 müssen
                                ; ausgeglichen werden.

        rte                     ; (Return from Exception)
```

Programm 8.5: Gebrauch des tc_TrapCode-Eintrags

Wie man sieht, benötigt man keine Exec-Funktion, um die Task-Exceptions zu benutzen. Dennoch gibt es auch hierfür zwei Routinen, die speziell für Traps bereitgestellt werden. Sie dienen dazu, den Eintrag tc_TrapAlloc/Free zu verändern, der jedoch nicht unbedingt berücksichtigt werden muß. Vollständigkeitshalber sollen auch diese beiden Funktionen hier aufgeführt werden:

AllocTrap	=	**-342 (Exec-Library)**

trapNum **d0** < Nummer des zu belegenden Trap-Befehls (0-15) oder -1, wenn der nächste freie Trap-Befehl belegt werden soll.

trapNum **d0** > Zurückgegeben wird die Nummer des Trap-Befehls (0-15), die belegt worden ist.

Erklärung Durch AllocTrap wird in der Task-Stuktur vermerkt, welche Trap-Befehle belegt sind. Diese Maske hat keine Auswirkung auf den Aufruf der Trap-Routine und dient nur zur Verwaltung der Traps.

FreeTrap	=	**-348 (Exec-Library)**

trapNum **d0** < Nummer des Trap-Befehls (0-15), der freigegeben werden soll.

Erklärung Der angegebene Trap-Befehl wird wieder freigegeben.

Will man die Exec-Routine, welche die Verwaltung der Traps übernimmt, umgehen, so kann man die Adresse der Trap-Routine direkt an die entsprechende Speicherstelle des Trap-Vektors schreiben. Dies hat jedoch zur Folge, daß auch interne Exceptions, die von einem anderen Programm ausgelöst wurden, diese Routine anspringen.

8.4 Das Message-System

Durch die Signal-Funktionen ist zwar ein gewisses Maß an Kommunikation möglich, doch der Austausch komplexer Daten kann dabei nicht stattfinden. Zur Erweiterung dient das Message-System.

Das Message-System besteht aus zwei Strukturen: dem Message-Port, einer Art Briefkasten, und der Message selbst, welche den Brief darstellt.

In der Intuition-Library haben wir dieses Nachrichtensystem schon einmal kurz kennengelernt, als wir auf eine IntuiMessage (das ist eine Art der Message-Struktur) gewartet haben. Aus dieser Struktur konnten wir dann die Ereignisse auslesen, die Intuition für uns vom Benutzer erhalten hat. Dabei hat Intuition diese Nachricht an den "Briefkasten" (MsgPort) unseres Fensters geschickt, welcher beim Öffnen generiert worden ist.

8.4.1 Die Message-Ports

Da wir uns jetzt einen eigenen MessagePort erstellen wollen, sollten wir uns zuerst die MessagePort-Struktur ansehen.

MsgPort-Struktur:

```
00      dc.l    *mp_Succ        ;┐
04      dc.l    *mp_Pred        ;│
08      dc.b     mp_Type        ;│ Node-Struktur
09      dc.b     mp_Pri         ;│
10      dc.l    *mp_Name        ;┘

14      dc.b     mp_Flags       ; Reaktionsflags
15      dc.b     mp_SigBit      ; Signalbits
16      dc.l    *mp_SigTask     ; Zeiger auf Task o. Int-Str.
20      ds.l     mp_MsgList     ; MsgList-Header = 14 Bytes
24               mp_SIZEOF
```

***mp_Succ, *mp_Pred, mp_Type, mp_Pri, *mp_Name**
Um die MessagePort-Strukturen in Listen verwalten zu können, beginnen sie mit einer Node-Struktur. Der Typ des Nodes muß als nt_MsgPort (4) angegeben werden. Der Port kann, muß aber nicht, in die Public-Port Liste von Exec aufgenommen werden.

mp_Flags
Mit dem Eintrag mp_Flags wird die Reaktion bestimmt, die die ankommende Nachricht auslösen soll. Hier sind folgende Flags vorbestimmt:

Name	Wert	Bedeutung
pa_Signal	00	Task durch Signale benachrichtigen

pa_SoftInt	01	Software-Interrupt auslösen
pa_Ignore	02	keine Reaktion
pf_Action	03	Routine (mp_SigTask) ausführen

pa_Signal Wenn eine Nachricht empfangen worden ist, werden die Signalbits aus dem Eintrag mp_SigBits mittels der Funktion Signal an den Task (mp_SigTask) gesendet. Diese Flags werden dann in den Eintrag tc_SigRecvd der Task-Struktur eingetragen.

pa_SoftInt Ist das Flag pa_SoftInt gesetzt, so wird ein Software-Interrupt mittels der Funktion Cause ausgelöst. Dabei wird der Zeiger auf die Interrupt-Struktur, die die Cause-Funktion benötigt, im Eintrag mp_SigTask erwartet.

pa_Ignore Wenn keine Reaktion ausgelöst werden soll, so muß das Flag pa_Ignore gesetzt werden. Dann wird dem Task nicht mitgeteilt, daß eine Nachricht vorliegt.

pf_Action Der Eintrag mp_SigTask wird als Zeiger auf eine Routine interpretiert, die beim Empfang einer Nachricht ausgeführt wird.

mp_SigBit
In diesem Wert steht die Nummer des Signalbits, welche dem Task bei einer empfangenen Nachricht gesendet werden soll.

***mp_SigTask**
Wie schon bei den mp_Flags angedeutet, kann der Eintrag *mp_SigTask drei verschiedene Bedeutungen haben. Dies ist abhängig von den gesetzten Flags.

pa_Signal
Bei der Flag-Kombination für pa_Signal wird das im Eintrag mp_SigBit abgelegte SignalBit an den Task gesendet, dessen Adresse (es handelt sich hierbei um die Adresse der Task-Struktur) hier abgelegt wurde.

pa_SoftInt
Sollten die Flags für pa_SoftInt gesetzt worden sein, so wird ein Software-Interrupt ausgelöst. Der dabei benötigte Zeiger auf die SoftInt-Struktur wird hier, im *mp_SigTask-Eintrag, abgelegt.

pa_Action
Bei der letzten Möglichkeit muß der Eintrag *mp_SigTask die Adresse einer Routine enthalten, die bei einer empfangenen Meldung ausgeführt werden soll.

mp_MsgList
Die ankommenden Nachrichten werden in einer Liste aufgereiht. Der Listenkopf ist an dieser Stelle in die MsgPort-Struktur eingebettet.

Erstellen und Entfernen von MsgPorts

Um nun einen solchen MessagePort für einen Task einzurichten, ist es günstig, sich eine kleine Routine zu basteln, die diese Arbeit erledigt. Auch die Runtime-Libraries vieler C-Compiler stellen dem Programmierer eine solche Funktion zur Verfügung. In Anlehnung an diese Funktionen haben wir die beiden Routinen zum Erstellen und Auflösen eines Ports Create- und DeletePort genannt. Wir benötigen dabei zwei weitere Exec-Funktionen, AddPort und RemPort. Diese beiden Funktionen sind dazu da, eine Port-Struktur in die PublicPort-Liste von Exec einzuhängen bzw. zu entfernen.

AddPort	=	**-354 (Exec-Library)**

***Port** a1 < Zeiger auf den Port, der in die Publi-Port-Liste von Exec eingefügt werden soll.

Erklärung Fügt einen Port in die von Exec verwaltete PublicPort-Liste ein, die von allen Tasks benutzt werden kann.

RemPort	=	**-360 (Exec-Library)**

***Port** a1 < Zeiger auf Port, der aus der PublicPort-Liste ausgeschlossen werden soll.

Erklärung RemPort dient dazu einen Port aus der öffentlichen Port-Liste zu entnehmen.

Man könnte natürlich auch einen Port erstellen, ohne ihn in die PublicPort-Liste von Exec einzutragen. Doch kann dann der sendende Task nicht mittels der FindPort-Funktion die Adresse unseres Message-Ports erfahren (näheres zu FindPort gleich)!

Jetzt aber wollen wir die beiden Funktionen zum Erstellen und Auflösen eines Message-Ports erläutern. Zunächst den Aufruf der beiden Funktionen:

```
...
move.l  #MsgPort_Name,a0     ; Zeiger auf Port-Namen nach a0
move.l  #0,d0           ; Priorität = Null
bsr     CreatePort      ; Port erstellen
tst.l   d0              ; konnte Port angelegt werden?
beq     Ende            ; nein, dann Ende
move.l  d0,MsgPort      ; Adresse speichern
...
...
```

```
        move.l   MsgPort,a0       ; Zeiger auf Port nach
        jsr      DeletePort       ; a0 und löschen
        ...
```

Nun folgen die Routinen Create- und DeletePort, die selbständig einen Message-Port einrichten und wieder entfernen können. Lediglich der Name des Ports sowie seine Priorität werden benötigt.

```
*
* Kapitel 8
* Create- und DeletePort-Funktionen
*

; Offsets für die Funktionen Create- und DeletePort

ExecBase      =       4
AddPort       =       -354
RemPort       =       -360
AllocSignal   =       -330
FreeSignal    =       -336
FindPort      =       -390
AllocMem      =       -198
FreeMem       =       -210

*
*       CreatePort
*
* a0 < Zeiger auf Port-Namen
* d0 < Priorität des Ports
*
* d0 > Zeiger auf erstellte Port-Struktur
*

CreatePort:
        movem.l  d1-d3/a1-a3/a6,-(a7)  ; Register retten
        move.l   ExecBase,a6
        move.l   d0,d3
        move.l   a0,a2            ; Parameter sichern

        move.l   a0,a1
        jsr      FindPort(a6)     ; Kontrollieren, ob es einen
        tst.l    d0               ; Port mit dem angegegebenen Namen
        bne      CPError          ; schon gibt

        move.l   #34,d0
        move.l   #$10000,d1
        jsr      AllocMem(a6)     ; Speicher für MsgPort-Struktur
        move.l   d0,a3            ; belegen
        beq      CPError

        move.l   #-1,d0
```

```
        jsr      AllocSignal(a6)       ; Signal belegen
        tst.b    d0
        bmi      DPFreeMem
                                 ; MsgPort-Struktur einrichten
        move.b   #4,8(a3)        ; mp_Type
        move.b   d3,9(a3)        ; mp_Pri
        move.b   #0,14(a3)       ;
        move.b   d0,15(a3)       ; mp_Sig
        move.l   a2,10(a3)       ; mp_Name
        move.l   276(a6),16(a3)  ; mp_SigTask

        move.l   a3,a1           ; Wenn ein Namen angegeben wurde,
        tst.l    10(a1)          ; wird der Port in die PublicPort-
        beq      CPBranch        ; Liste aufgenommen
        jsr      AddPort(a6)
        bra      CPOK

CPBranch:
        lea      20(a3),a1       ; Sollte der Port nicht in die PP-
        move.l   a1,(a1)         ; Liste aufgenommen werden, muß man
        addq.l   #4,(a1)         ; den Listenkopf der Nachrichten
        clr.l    4(a1)           ; (MsgList) von "Hand" einrichten
        move.l   a1,8(a1)        ; !!! Sonst hätte das AddPort für
                                 ; uns erledigt !

CPOK:   move.l   a3,d0           ; Zeiger auf Port "bergeben
CPError:
        movem.l  (a7)+,d1-d3/a1-a3/a6   ; Pop
        rts

*
*       DeletePort
*
* a0 < Zeiger auf Port-Struktur, die aufgelöst werden soll
*

DeletePort:
        movem.l  d1-d3/a1-a3/a6,-(a7)
        move.l   ExecBase,a6
        move.l   a0,a3           ; Parameter retten

        move.l   a3,a1           ; Hat der Port einen Namen, so ist
        tst.l    10(a1)          ; er in die PP-Liste eingetragen
        beq      DPBranch        ; worden. Jetzt müssen wir ihn
        jsr      RemPort(a6)     ; wieder entfernen.
DPBranch:

        move.b   15(a3),d0       ; Signal freigeben
        ext.w    d0
        ext.l    d0
        jsr      FreeSignal(a6)

DPFreeMem:
        move.l   #34,d0
        move.l   a3,a1
        jsr      FreeMem(a6)     ; Speicher freigeben
```

```
        movem.l  (a7)+,d1-d3/a1-a3/a6
        moveq    #0,d0          ; Null übergeben
        rts
```

Programm 8.6: CreatePort und DeletePort

8.4.2 Die Messages

Nachdem wir die beiden Routinen zum Einrichten und zum Löschen eines MessagePorts kennengelernt haben, wollen wir uns jetzt um die "Post" kümmern, die wir hoffentlich in unserem "Briefkasten" finden werden. Hierzu müssen wir uns die Message-Struktur ansehen.

Message-Struktur:

```
00    dc.l   *mn_Succ          ;┐
04    dc.l   *mn_Pred          ;│
08    dc.b    mn_Type          ;│ Node-Struktur
09    dc.b    mn_Pri           ;│
10    dc.l   *mn_Name          ;┘

14    dc.l   *mn_ReplyPort     ; Zeiger auf Antwort-Port
18    dc.w    mn_Length        ; Länge der folgenden Daten
20            mn_SIZEOF
```

***mn_Succ, *mn_Pred, mn_Type, mn_Pri, *mn_Name**
Zunächst besteht die Message-Struktur aus einer Node-Struktur, die benutzt wird, um alle ankommenden Nachrichten zu verwalten. Das Abarbeiten der Meldungen geschieht dann nach dem sogenannnten first-in-first-out-Prinzip. Das heißt: Die Message, die zuerst gesendet wurde, wird auch zuerst behandelt.

***mn_ReplyPort**
Nach der Knoten-Struktur folgt ein Zeiger, der auf einen ReplyPort verweist. Dieser Port muß/kann eine Bestätigung des Empfangs erhalten.

mn_Length
Die nachfolgenden Daten können je nach Art der Nachricht unterschiedlich ausfallen. Es muß lediglich an dieser Stelle die Länge der Daten und die Länge der Struktur mitgeteilt werden.

Abschicken und Empfangen von Messages

Jetzt, nachdem wir die Message-Struktur kennengelernt haben, können wir Nachrichten verschicken. Dazu benötigt man einen Brief in Form einer Message-Struktur und die Adresse des Empfängers, sprich die Adresse des MsgPorts, dem man eine Nachricht senden will. Um die Adresse eines Message-Ports zu bekommen, können wir mittels der FindPort-Funktion die Pu-

blicPort-Liste von Exec nach einem Namen durchsuchen lassen. Deshalb sollte man, wie schon erwähnt, seinen Port stets in die PublicPort-Liste eintragen (AddPort). Außerdem sollte man darauf achten, daß man nicht zwei Ports mit dem gleichen Namen anlegt! Aber um diese Kleinigkeiten müssen wir uns, dank der CreatePort und DeletePort Funktionen, keine Gedanken mehr machen.

Wie gesagt, benötigen wir die Funktion FindPort, um die Adresse eines in der PublicPort-Liste eingetragenen Ports zu ermitteln.

FindPort	=	**-390 (Exec-Library)**

***Name**	a1	<	Zeiger auf eine, mit Null abgeschlossene Zeichenkette.
Port	d0	<	Adresse der gesuchten Port-Struktur oder, wenn die Struktur nicht gefunden werden konnte, eine Null.
Erklärung			Mit der Funktion FindPort kann man in der PublicPort-Liste von Exec nach einer, durch den Namen bestimmten, Port-Struktur suchen lassen. Man erhält entweder die gesuchte Adresse oder eine Null, wenn kein Port dieses Namens gefunden werden konnte.

Nebenbei sei noch bemerkt, daß die FindPort-Funktion lediglich den Zeiger auf den Listenkopf der PublicPort-Liste ins Adreßregister 0 legt und dann die Funktion FindName aufruft (die Listenköpfe der PublicPort-Liste und aller anderen Systemlisten befinden sich im Datenbereich der Exec-Library).

Haben wir die Adresse des Message-Ports gefunden, können wir nun die Meldung senden. Hierzu dient die Funktion PutMsg.

PutMsg	=	**-366 (Exec-Library)**

***Port**	a0	<	Zeiger auf den Port, dem eine Nachricht gesendet werden soll.
***Message**	a1	<	Zeiger auf die Message-Struktur, die an den Port gesendet werden soll.
Erklärung			PutMsg erlaubt es, Meldungen an einen Message-Port zu verschicken. Dabei wird die Message-Struktur an die Liste der Nachrichten des Message-Ports gehängt. Die weitere Arbeit der PutMsg-Funktion ist von der, im Eintrag mp_Flags gesetzten Bit-Kombination abhängig.

Um die Verbindung mit dem Signal-System deutlich zu machen folgt nun ein Auszug aus der PutMsg-Funktion des Kick-Start-ROM V2.0.

```
PutMsg: ; Exec-Library (-366)
        ; a0 < Zeiger auf Port
        ; a1 < Zeiger auf Message-Struktur

        MOVEQ    #5,D0            ; Typ = nt_Message = 5
        MOVE.L   A0,D1            ; Zeiger auf Port nach d1
LF820C0 LEA      $14(A0),A0       ; Adresse der MsgList-Header

        MOVE.W   #$4000,$DFF09A  ; Interrupts sperren
        ADDQ.B   #1,$126(A6)      ; IDNestCnt++ (Disable)
        MOVE.B   D0,8(A1)         ; Typ eintragen
        ADDQ.L   #4,A0            ; a0 += 4
        MOVE.L   4(A0),D0         ; Zeiger auf Endknoten
        MOVE.L   A1,4(A0)         ; neue Message einbinden
        MOVE.L   A0,(A1)          ; Adresse von lh_Tail in den
                                  ; lh_Succ-Eintrag der Message-
                                  ; Struktur einsetzen
        MOVE.L   D0,4(A1)         ; alter Endknoten nach lh_Pred
        MOVEA.L  D0,A0            ; Endknotenadresse nach a0
        MOVE.L   A1,(A0)          ; lh_Succ-Eintrag des alten
                                  ; Endknoten auf neue Nachricht
                                  ; setzen

        ; Nachdem die Nachricht in die MsgList des MsgPorts
        ; eingefügt worden ist, wird nun auf die Behandlung der
        ; Nachricht eingegangen!

        MOVEA.L  D1,A1            ; Zeiger auf Port nach a1
        MOVE.L   $10(A1),D1       ; mp_SigTask nach d1
        BEQ.S    LF82106          ; kein SigTask eingetragen

        MOVE.B   $E(A1),D0        ; mp_Flags auslesen
        ANDI.W   #3,D0            ; obere Bits ausblenden
        BEQ.S    LF82122          ; d0 = pa_Signal ?

        CMPI.B   #1,D0            ; testen ob d0 = pa_SoftInt
        BNE.S    LF82116          ; nein, dann weiter

pa_SoftInt:                       ; Routine für SoftInterrupt
        MOVEA.L  D1,A1            ; mp_SigTask (Zeiger auf SoftInt)
        JSR      -$B4(A6)         ; nach a1 und SoftInt auslösen
                                  ; (Cause)

LF82106
pa_Ignore:                        ; Routine für "Ignore"
        SUBQ.B   #1,$126(A6)      ; IDNestCnt-- (Enable)
        BGE.S    LF82114          ; Interrupts wirklich freigeben ?
        MOVE.W   #$C000,$DFF09A  ; Interrupts freigeben
LF82114 RTS                       ; Ende der PutMsg-Funktion
```

```
LF82116 CMPI.B   #2,D0            ; testen ob d0 = pa_Ignore, wenn ja
        BEQ.S    LF82106          ; dann nach pa_Ignore verzweigen

pf_Action:                        ; Routine für Action-Flag
        MOVEA.L  D1,A0            ; Wert von mp_SigTask nach a0

        ; Der Zeiger auf die Routine, die ausgeführt werden soll
        ; steht nun im Adreßregister a0

        JSR      (A0)             ; Routine ausführen
        BRA.S    LF82106          ; >Ende< (pa_Ignore)

LF82122 MOVE.B   $F(A1),D0        ; mp_SigBits auslesen
        ADDQ.B   #1,$127(A6)      ; TaskDisableNestingCounter++
        SUBQ.B   #1,$126(A6)      ; InterruptDisableNestingCounter--
        BGE.S    LF82138          ; Interrupts wirklich freigeben ?

        MOVE.W   #$C000,$DFF09A   ; Interrupts freigeben

LF82138 MOVEA.L  D1,A1            ; Zeiger auf Port nach a1
        MOVEQ    #0,D1            ; d1 löschen
        BSET     D0,D1            ; SignalBits in d1 setzen
        MOVE.L   D1,D0            ; und nach d0 schieben
        JSR      -$144(A6)        ; Signal abschicken (Signal)
        BRA      $F82578          ; (Permit) > Ende
```

Bild 8.3: ROM-Auszug der PutMsg-Routine

Wie man sieht, wird die angegebene Nachricht an die Message-Liste des Empfänger-Ports gehängt. Danach wird, wie durch die Flag-Kombination vom mp_Flags bestimmt, reagiert.

Als nächstes müssen wir uns um die "Briefkastenleerung" kümmern. Sie wird mit Hilfe des Befehls GetMsg bewerkstelligt.

GetMsg	=	**-372 (Exec-Library)**

***Port**	**a0**	<	Zeiger auf eine MessagePort-Struktur, aus der eine Nachricht entnommen werden soll.
***Message**	**d0**	>	Nachdem die Funktion-GetMsg ausgeführt wurde, erhält man die Adresse auf die nächste Message-Struktur. Wenn keine Nachricht mehr vorliegt, wird eine Null zurückgegeben.
Erklärung			Durch GetMsg kann man die nächste Nachricht (es wird nach dem first-in-first-out-System vorgegangen) entnehmen.

Um die Nachricht abzuholen, wird der Zeiger auf einen MessagePort benötigt, von dem die nächste Nachricht übernommen werden soll. Nachdem die Meldung aus der Liste entfernt worden ist, erhält man die Adresse auf die Message-Struktur zurück. Sollte keine Nachricht anliegen, erhält man eine Null.

Auch hierzu ein Auszug aus dem KickStart-ROM (2.0)

```
GetMsg: ; GetMsg (-372)
        ; a0 < Zeiger auf Ports

        LEA      $14(A0),A0      ; Adresse des mp_MemList Eintrags
        MOVE.W   #$4000,$DFF09A ; Interrupts sperren
        ADDQ.B   #1,$126(A6)    ; IDNestCnt++ (Disable)

        MOVEA.L  (A0),A1        ; Zeiger auf erste Nachricht
        MOVE.L   (A1),D0        ; ln_Succ-Eintrag auslesen
        BEQ.S    LF82166        ; Null ? Dann war die Liste leer
        MOVE.L   D0,(A0)        ; sonst Succ als erstes eintragen
        EXG      D0,A1          ; Registerinhalte austauschen
        MOVE.L   A0,4(A1)       ; Vorgänger (ln_Pred) setzen

LF82166 SUBQ.B   #1,$126(A6)    ; IDNestCnt-- (Enable)
        BGE.S    $F82174        ; Interrupts wirklich freigeben ?
        MOVE.W   #$C000,$DFF09A ; Interrupt freigeben
        RTS                     ; Ende
```

Bild 8.4: ROM-Auszug der GetMsg-Routine

Haben wir die Meldung vom Port übernommen, können wir auf die Message-Daten zugreifen. Betrachtet man den Vorgang ganz neutral, so stellt man fest, daß es sich bei einer Nachricht um einen Speicherbereich handelt, auf den ein zweiter Task für einen bestimmten Zeitraum zugreifen kann. Die Länge kann vom Empfängertask festgelegt werden. Während dieser Zeit sollte der sendende Task nicht auf den Datenbereich schreibend (!) zugreifen. Wird die Nachricht mittels der Funktion ReplyPort vom Empfängertask beantwortet, hat der Task, von dem die Nachricht stammt, wieder die Erlaubnis, auf seine Daten zuzugreifen.

ReplyMsg	=	**-378 (Exec-Library)**

***Message** a1 < Zeiger auf die Message-Struktur, deren Empfang bestätigt werden soll.

Erklärung Mit der Funktion ReplyMsg kann man den Empfang einer Meldung bestätigen. Dies kann wichtig sein, da einige Tasks auf eine Rückmeldung warten, bevor sie ihre Arbeit fortführen. Durch den Aufruf wird der Typ des Knotens der Message-Struktur

auf 7, also nt_ReplyMsg, geändert. Danach wird sie an den angegebenen ReplyPort, dessen Adresse in der Struktur enthalten ist, zurückgesendet.

Auch hier wollen wir uns die Routine näher ansehen:

```
ReplyMsg:; Exec-Library (-378)
        ; a1 < Zeiger auf Message die bestätigt werden soll

        MOVEQ    #7,D0          ; Typ = nt_ReplyMsg (7)
        MOVE.L   $E(A1),D1      ; Zeiger auf ReplyPort nach d1
        MOVEA.L  D1,A0          ; testen ob ReplyPort eingetragen
        BNE.S    LF820C0        ; ist. Wenn ja, dann zurück senden
        MOVE.B   #6,8(A1)       ; sonst Message in nt_FreeMsg (6)
        RTS                     ; ändern

        ; PutMsg-Funktion:

PutMsg: MOVEQ    #5,D0          ;
        MOVE.L   A0,D1          ;
LF820C0 LEA      $14(A0),A0     ; ab hier wird die Nachricht
        MOVE.W   #$4000,$DFF09A ; zurückgesendet
        ADDQ.B   #1,$126(A6)    ;
        ...
```

Bild 8.5: ROM-Auszug der ReplyMsg-Routine

Wie erwähnt, wird zusätzlich zu der GetMsg-Funktion auch noch die ReplyMsg-Funktion benötigt. Manche Tasks warten sogar solange, bis sie auf ihre Nachricht eine Antwort erhalten haben. Meist werten sie dann die zurückerhaltene Nachricht aus, falls der empfangende Task neue Daten in die Message-Struktur geschrieben hat.

Warten auf Messages

Zum Schluß müssen wir uns noch überlegen, wie wir auf eine Nachricht warten können. Die erste Möglichkeit besteht darin, daß man immer wieder versucht, eine Nachricht vom MessagePort zu bekommen. Bei dieser Möglichkeit muß unser Task ständig bearbeitet werden und nimmt so den anderen Tasks Rechenzeit weg. Um dies zu verhindern, kann man die Funktion WaitPort einsetzen.

WaitPort	=	**-384 (Exec-Library)**

***Port** **a0** < Zeiger auf eine Port-Struktur

Erklärung Mit der Funktion WaitPort wird solange gewartet, bis der angegebenen Port eine Nachricht empfängt.

Wiederum wollen wir uns ein ROM-Auszug ansehen, der die Routine verständlicher machen soll:

```
WaitPort:; Exec-Library (-384)
        ; a0 < Zeiger auf Port-Struktur

        MOVEA.L  $14(A0),A1      ; Zeiger auf erste Nachricht
        TST.L    (A1)            ; laden und testen. Wenn eine
        BNE.S    LF8219A         ; Meldung anliegt zurück.

        MOVE.B   $F(A0),D1       ; mp_SigBits aus Port holen
        LEA      $14(A0),A0      ; mp_MsgList der Nachrichten
        MOVEQ    #0,D0           ; d0 löschen
        BSET     D1,D0           ; Bits übertragen

        MOVE.L   A2,-(A7)        ; Inhalt von a2 retten
        MOVEA.L  A0,A2           ;

LF8218E JSR      -$13E(A6)       ; (Wait)
        MOVEA.L  (A2),A1         ; Message nach a1
        TST.L    (A1)            ; ist etwas angekommen ?
        BEQ.S    LF8218E         ; nein, dann wiederum warten!

        MOVEA.L  (A7)+,A2        ; Inhalt von a2 restaurieren

LF8219A MOVE.L   A1,D0           ; angekommene Nachricht nach
        RTS                      ; d0 und Funktion beenden!
```

Bild 8.6: ROM-Auszug der WaitPort-Routine

Das Programm wird durch die WaitPort-Funktion solange gestoppt, bis die SignalBits des MessagePorts empfangen werden. Die Routine hat dann folgendes Aussehen:

```
        ...
MsgLoop:
        move.l   4,a6            ; ExecBase nach a6
        move.l   Port,a0         ; Nachricht vom angegebenen
        jsr      GetMsg(a6)      ; Port abholen.
        move.l   d0,Message      ; Zeiger speichern
        bne      CheckMessage    ; verzweigen wenn wir eine
                                 ; Meldung empfangen haben
```

```
        move.l  Port,a1         ; sollte keine Nachricht an-
        jsr     WaitPort(a6)    ; gekommen sein, müssen wir
        bra     MsgLoop         ; warten!
        ...
```

Bild 8.7: Warten auf eine Nachricht mit WaitPort

Eine weitere Möglichkeit besteht darin, daß man die Exec-Funktion Wait direkt benutzt. Auch hier wird der Task in die TaskWait-Liste übertragen. Erst wenn eins der angegebenen Signalbits empfangen wurde, wird er wieder in den Ready-Zustand versetzt.

```
        ...
MsgLoop:
        move.l  4,a6            ; ExecBase nach a6
        move.l  Port,a0         ; Nachricht vom angegebenen
        jsr     GetMsg(a6)      ; Port abholen.
        move.l  d0,Message      ; Zeiger speichern
        bne     CheckMessage    ; verzweigen wenn wir eine
                                ; Meldung empfangen haben.
                                ; Ist keine Nachricht vorhan-
                                ; den, wird auf die in der MsgPort-
        move.l  Signal,d0       ; Struktur abgelegten Signal-Bits
        jsr     Wait(a6)        ; gewartet.
        bra     MsgLoop         ;
        ...
```

Bild 8.8: Warten auf eine Nachricht mit Wait

Der Unterschied zu der WaitPort-Funktion besteht darin, daß bei letzterer nur auf einen Port, also nur auf ein Signal-Bit, geachtet werden kann. Bei Wait hingegen kann auf mehrere Signale, also auch auf mehrere MessagePorts, reagiert werden. So wird es möglich z.B. gleichzeitig auf Ereignisse von zwei verschiedenen Fenstern zu warten.

8.4.3 Demoprogramm

Den Schluß dieses Kapitels bilden zwei Programme, welche die Funktion des Message-Systems demonstrieren sollen. Der erste Task richtet einen Message-Port ein und wartet dann auf eine Nachricht. Erhält er eine solche, gibt er den mit der Nachrichten-Struktur gesendeten Text auf seinem Fenster aus.

```
*
* Kapitel 8
* Demonstrationsprogramm #1 für das Message-System
*
```

```
ExecBase        =       4

; Weitere Offsets ...

Start:
        ; ...

        ; Zunächst wird die Dos-Library und ein "einfaches"
        ; Fenster geöffnet, über welches die Kommunikation
        ; ablaufen soll.

        ; ...

        lea       Text1,a0
        bsr       MyPrint         ; Text ausgeben

        move.l    #MsgPort_Name,a0
        move.l    #0,d0
        bsr       CreatePort      ; Msg-Port anlegen
        move.l    d0,MsgPort

        move.l    ExecBase,a6
WaitLoop:
        lea       Text2,a0        ; Text ausgeben
        bsr       MyPrint

        move.l    MsgPort,a0
        jsr       WaitPort(a6)    ; auf Nachricht warten

        move.l    MsgPort,a0
        jsr       GetMsg(a6)      ; Nachricht abholen

        move.l    d0,Message
        move.l    d0,a0
        tst.w     18(a0)          ; Länge = 0?
        bne       Weiter          ; Nein, dann weiter

        lea       Text4,a0
        bsr       MyPrint         ; Ende-Text ausgeben

        pea       Ende            ; Rücksprungadresse = Ende
        bra       Schluß          ; Nachricht bestätigen

Weiter: lea       Text3,a0
        bsr       MyPrint         ; Empfangsbestätigung ausgeben

        move.l    Message,a0
        add.l     #20,a0
        bsr       MyPrint         ; Nachricht ausgeben

        pea       WaitLoop        ; Adresse für Loop setzen

Schluß: move.l    Message,a1
        jsr       ReplyMsg(a6)    ; Nachricht bestätigen
        rts                       ; Loop verzweigen
```

```
Ende:   move.l   MsgPort,a0        ; Port entfernen
        jsr      DeletePort

        move.l   DosBase,a6
        move.l   WindowHD,d1
        jsr      Close(a6)         ; Fenster schließen
WinError:

        move.l   ExecBase,a6
        move.l   DosBase,a1
        jsr      CloseLib(a6)      ; Dos-Lib schließen

Error:  rts                        ; Programm beenden

        ; ...
        ; Ausgabe-Routine
        ; Datenbereich
        ; ...
```

Programm 8.7: Message-Demonstration, Programm 1

Das zweite Programm liest eine Textzeile ein und sendet sie mittels PutMsg an den Message-Port des ersten Tasks, der seinerseits die Nachricht auswertet. Danach wartet er auf die Bestätigung der Meldung, wonach er wiederum die Eingabe einer Zeichenkette erlaubt.

```
*
* Kapitel 8
* Demonstrationsprogramm #2 für das Message-System
*

ExecBase      =       4

; Weitere Offsets ...

Start:
        ; ...

        ; Zunächst wird die Dos-Library und ein "einfaches"
        ; Fenster geöffnet, über welches die Kommunikation
        ; ablaufen soll.

        ; ...

        lea      Text1,a0          ; Hallo-Text ausgeben
        bsr      MyPrint

        move.l   ExecBase,a6

        lea      YourPortName,a1         ; Adresse des Message-Ports des
        jsr      FindPort(a6)      ; ersten Tasks suchen.
        move.l   d0,YourPort       ;
```

```
        beq      Exit

        sub.l    a0,a0                   ; Name ist belanglos
        move.l   #0,d0                   ; Priorität = 0
        bsr      CreatePort      ; Reply-Port einrichten
        move.l   d0,MsgPort

        move.l   ExecBase,a6
        move.l   MsgPort,mn_ReplyPort    ; ReplyPort einsetzen

MsgLoop:
        lea      Text2,a0        ; Eingabeaufforderung ausgeben
        bsr      MyPrint

        move.l   #mn_Data,d2
        move.l   #100,d3
        bsr      M               bsr      MyReadLN        ; String einlesen
        move.w   d0,mn_Length

        pea      MsgLoop         ; Stack mit MsgLoop belegen
        tst.l    d0
        bne      Weiter                  ; Keine Eingabe
        move.l   #Ende,(a7)      ; Ja, dann Ende auf Stack ablegen

Weiter:
        lea      Text3,a0        ; Sende-Text ausgeben
        bsr      MyPrint

        move.l   YourPort,a0
        lea      MyMsg,a1
        jsr      PutMsg(a6)      ; Nachricht senden

        lea      Text4,a0        ; Warte-Text ausgeben
        bsr      MyPrint

        move.l   MsgPort,a0
        jsr      WaitPort(a6)    ; auf Bestätigung warten

        move.l   MsgPort,a0      ; und Message aus ReplyPort
        jsr      GetMsg(a6)      ; entfernen
        rts                      ; Programm beenden oder wiederholen. Je
                                 ; nachdem, welche Rücksprungadresse auf
                                 ; dem Stack abgelegt worden ist.

Ende:   move.l   MsgPort,a0
        bsr      DeletePort      ; ReplyPort "abbauen"

Remove:
        move.l   DosBase,a6
        move.l   WindowHD,d1
        jsr      Close(a6)       ; Fenster schließen
WinError:

        move.l   ExecBase,a6
        move.l   DosBase,a1
        jsr      CloseLib(a6)    ; Dos-Library schließen
```

```
Error:  rts                                 ; Programm beenden

Exit:   lea       ErrText1,a0       ; Fehler-Text ausgeben
        bsr       MyPrint

        move.l    #ErrText1,d2
        move.l    #2,d3
        pea       Remove            ; Rücksprungadresse festlegen
        bra       MyReadLN          ; Text einlesen (Repeat until
                                    ; Keypressed)

        ; ...
        ; Ein- und Ausgaberoutinen
        ; ...
        ; Datenbereich

MyMsg:            dc.l      0,0       ; mn_Succ, mn_Pred
                  dc.b      0,0       ; mn_Typ, mn_Pri
                  dc.l      0         ; mn_Name

mn_ReplyPort      dc.l      0         ; mn_ReplyPort
mn_Length                   dc.w    0   ; mn_Length
mn_Data           ds.l      100       ; Zeichenkettenpuffer
```

Programm 8.8: Message-Demonstration, Programm 2

8.5 Libraries (Bibliotheken)

Bisher haben wir schon einige Libraries kennengelernt und uns aus ihrem reichhaltigen Angebot an Funktionen bedient. Über den Aufbau haben wir aber nur einen kleinen Einblick erhalten, der in diesem Kapitel vertieft werden soll.

Im Grunde ist eine Library nichts anderes als eine Sammlung von einzelnen Routinen, auf die ein Programmierer zurückgreifen kann. Jede Library ist für ein spezielles Gebiet zuständig. So enthält die Graphics-Library z.B. Routinen, die für Grafikausgaben benutzt werden können.

Folgende Libraries stehen zur Verfügung:

D "arp.library"	Die AmigaDOS Replacement Project Library ist sehr weit verbreitet und enthält viele nützliche Funktionen wie z.B. einen FileRequester.
D "diskfont.library"	Laden und Suchen von Zeichensätzen, die auf Diskette vorliegen.
R "clist.library"	Die Funktionen der CList-Library werden zur Manipulation der Char-Listen benutzt.
R "console.library"	Die console.library beinhaltet zahlreiche Funktionen, die die Ein- und Ausgabe mit dem Console-Device unterstützen.

R **"dos.library"**	In der DOS-Library sind Funktionen zur Bearbeitung von Dateien, einfachen Fenstern und sonstigen Datenträger-Einrichtungen enthalten.
R **"exec.library"**	Die Exec-Library ist die Betriebs-System-Library. In ihr findet man viele Funktionen, die die Arbeit mit dem Betriebssystem erleichtern.
R **"graphics.library"**	Wie schon angesprochen, enthält die Graphics-Library eine große Anzahl von Funktionen, die für die Grafikausgabe benutzt werden können.
D **"icon.library"**	Die Icon-Library, die sich nicht im ROM befindet sondern von Diskette nachgeladen werden muß, ist für die Darstellung der Icons auf der Workbench zuständig.
R **"intuition.library"**	Die Funktionen der Intuition-Library stellen die grafische Benutzeroberfläche (Windows, Screens usw.) des Amiga dar.
D **"janus.library"**	Die Janus-Library bietet einige Funktionen für die Programmierung des Side-Car bzw. der Brückenkarte (PC-Emulator) des Amigas an.
D **"layers.library"**	Die Funktionen der Layers-Library werden von der Intuition- und der Graphics-Library benutzt, da sie für die Verwaltung von sich überlagernden Grafikausschnitten zuständig sind.
D **"mathffp.library"**	Diese Library enthält Funktionen für Rechnungen mit Fließkommazahlen.
D **"mathieeedoubbas.lib"**	Diese Library enthält mathematische Funktionen für Rechnungen mit doppelt-genauen Fließkommazahlen.
D **"mathtrans.library"**	Hier sind mathematische Sonderfunktionen wie sin, cos, tan usw. gesammelt.
D **"potgo.library"**	Programmierung der analogen Eingänge (Maus).
D **"timer.library"**	Zeitmessung durch CIAs.
D **"translator.library"**	Die Translator-Library enthält nur eine Funktion. Sie konvertiert einen Text in Lautschrift, um ihn durch das Narrator-Device ausgeben lassen zu können.

Wie man schon an dem großen Angebot an Libraries sehen kann, ist jeder Bereich des Computers mit hilfreichen Routinen ausgestattet. Dies erleichtert die Arbeit des Programmierers und verhindert Kompatibilitätsprobleme.

8.5.1 Aufbau einer Library

Doch jetzt wollen wir uns etwas näher mit dem Aufbau einer Library, die sich im Speicher befindet, beschäftigen. Grundsätzlich kann man sagen: Eine Library unterteilt sich in zwei wichtige Abschnitte. Zum einen in den Funktionsteil und zum anderen in die Library-Struktur mit der Sprungtabelle. Dabei sind die beiden Teile lediglich durch die Sprungtabelle verbunden.

Zu dem Funktionsteil braucht man wohl nichts mehr zu sagen. Er enthält abgeschlossene Routinen, die spezielle Aufgaben

übernehmen können. Viel interessanter ist die Sprungtabelle und die, sich direkt daran anschließende Library-Struktur. Sie werden durch die Basisadresse, die mit der Adresse des ersten Eintrags der Library-Struktur übereinstimmt, geteilt. Man erhält diese Adresse, wenn man die Library mittels der (Old)OpenLibrary-Funktion öffnet. Ausgehend von dieser Adresse kann man also sagen, daß die Sprungtabelle im negativen Bereich (adressierbar durch negative Offsets) und die Library-Struktur im positiven Bereich liegt. Diesen Aufbau soll die folgende Skizze verdeutlichen.

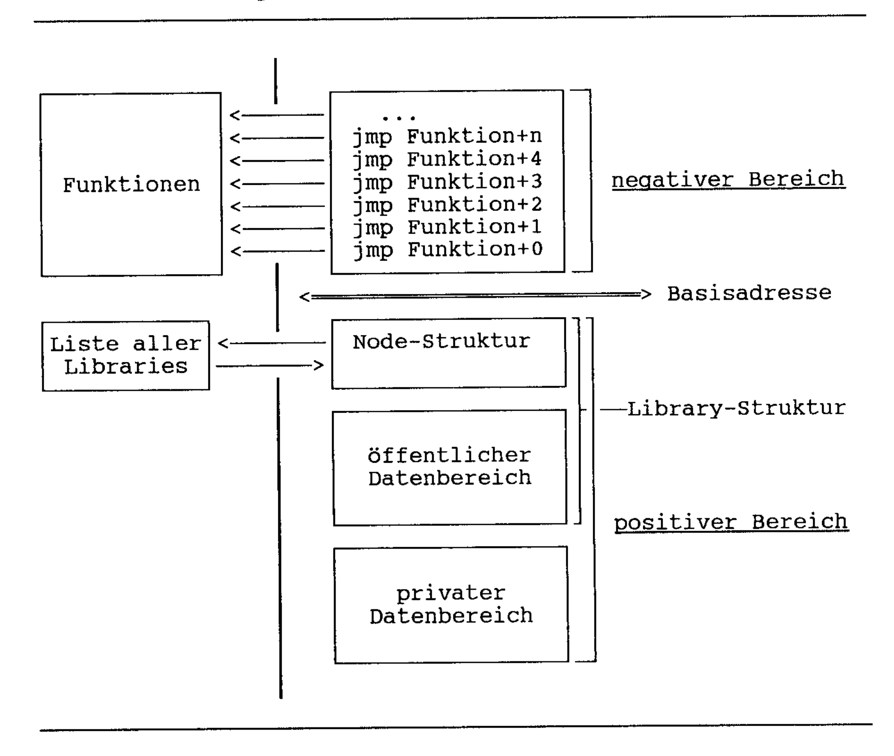
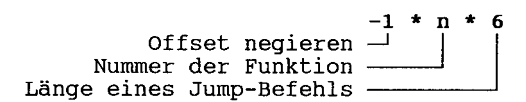


Bild 8.9: Skizze zum Aufbau einer Library

Wie man aus der Skizze erkennen kann, besteht die Sprungtabelle lediglich aus einer Anzahl von Jump-Befehlen, die in den Funktionsteil verzweigen. Um von der Basisadresse in den Bereich der Sprungtabelle zu gelangen, benötigt man einen negativen Offset, der ein Vielfaches von sechs ist. Dies ist begründet in der Länge eines Sprungbefehls OpCode (2 Byte) + Adresse (4 Byte) = Jump-Befehl (6 Byte).

```
          ...
-30       jmp              Funktion+n
-24       jmp              Funktion+4
-18       jmp              Funktion+3
-12       jmp              Funktion+2
-06       jmp              Funktion+1
 00       ...
```

Für die Offsetberechnung ergibt sich dann folgende Formel:



Im Programm sähe dies wie folgt aus:

```
...
move.l      4,a6
jsr         -1*n*6(a6)
...
```

Es ist noch wichtig zu wissen, daß man immer das Adreßregister a6 nehmen sollte, um in die Sprungtabelle einer Library zu verzweigen, da fast alle Funktionen in a6 die Adresse der Library-Basis erwarten.

Nachdem wir uns mit der Sprungtabelle auseinandergesetzt haben, schenken wir nun der Library-Struktur unsere Aufmerksamkeit. Sie enthält Daten, die zur Verwaltung der Library dienen.

```
jmp   ...              ; Sprungtabelle
jmp   ...
...
jmp   ...
```

Library-Struktur:

```
00    dc.l   *lib_Succ          ;┐      <====== Basisadresse
04    dc.l   *lib_Pred          ;│
08    dc.b    lib_Type          ;│  Node-Struktur
09    dc.b    lib_Pri           ;│
10    dc.l   *lib_Name          ;┘

14    dc.w    lib_Flags         ; Flags für Library-Struktur
16    dc.w    lib_NegSize       ; Länge der Sprungtabelle Byte
18    dc.w    lib_PosSize       ; Länge des Datenteils in Byte
20    dc.w    lib_Version       ; Versionsnummer der Library
22    dc.w    lib_Reversion     ; Revisionsnummer der Lib.
24    dc.l   *lib_IDString      ; Identifikationsstring
28    dc.l    lib_Sum           ; Prüfsumme d. Sprungtabelle
32    dc.w    lib_OpenCnt       ; Zähler der Benutzer

      ; Spezialdatenbereich der Library
      ...
```

***lib_Succ, *lib_Pred, lib_Type, lib_Pri, *lib_Node**
Im Anschluß an die Sprungtabelle finden wir wieder eine Node-Struktur, die es Exec ermöglicht, alle Libraries (die im Speicher vorliegen) zu verwalten. Die List-Struktur (Kopf der Liste) finden wir in der ExecBase ab dem Byte 378.

lib_Flags
Der Eintrag lib_Flags enthält den Zustand, in der sich die Library im Augenblick befindet. Dabei sind folgende Werte möglich:

Library-Flag	Wert	Bedeutung
libf_Summing	00	Checksumme wird gerade berechnet
libf_Changed	01	Library wurde geändert
libf_SumUsed	02	Kontrolle der Checksumme
libf_DelExp	03	Library wird aus dem System entfernt

lib_NegSize
Größe des negativen Datenbereichs. Hiermit ist die Länge der Sprungtabelle gemeint, die vor der Library-Struktur, also im negativen Bereich, liegt.

lib_PosSize
Größe des positiven Datenbereichs. Die Länge dieses Bereichs kann je nach Library unterschiedlich ausfallen. Dies liegt daran, daß neben den Standarddaten auch interne Daten abgelegt werden können.

lib_Version
Versionsnummer der Library.

lib_Reversion
Reversionsnummer (Überarbeitungsnummer) der Library.

lib_IDString
Zeiger auf eine Zeichenkette, welche den Identifikationstext enthält.

lib_Sum
Der Eintrag lib_Sum enthält die Prüfsumme über die Sprungtabelle. Wenn man einen Vektor der Tabelle "verbogen" (geändert) hat, sollte man diesen Wert neu berechnen (siehe auch SumLibrary).

lib_OpenCnt
Durch den Wert OpenCnt wird die Anzahl der Benutzer festgelegt, welche die Library geöffnet haben. Wenn kein Programm die entsprechende Library benötigt, kann sie aus der Liste der Libraries entfernt und dadurch der Speicherplatz freigegeben werden. Dies geschieht jedoch nur, wenn das Flag libf_DelExp (verzögertes Entfernen) im Eintrag lib_Flags gesetzt ist. Jetzt dürfte es auch klar sein, warum die Libraries immer geschlossen werden sollten, wenn man sie nicht mehr benutzt.

Datenbereich
Im Anschluß an die Library-Struktur folgt ein Datenteil, der von Library zu Library unterschielich ausfallen kann.

8.5.2 Routinen zur Library-Verwaltung

Nachdem wir uns die Library-Struktur angesehen haben, sollten wir uns den Funktionen der Library zuwenden. Sicherlich ist ihnen schon aufgefallen, daß alle Library-Offsets bei -30 beginnen. Das liegt daran, daß die ersten vier Funktionen (-6 bis -24) zur Verwaltung der Library benutzt werden. Sie haben folgende Aufgaben:

lib_Open	**-06**	Routine, die beim Öffnen der Library ausgeführt wird (wird von OpenLibrary angesprungen).
lib_Close	**-12**	Routine, die beim Schließen der Library ausgeführt wird (wird von CloseLibrary angesprungen).
lib_Expunge	**-18**	Routine, die beim Entfernen der Library aufgerufen wird (wird von RemLibrary angesprungen).
lib_Extfunct	**-24**	Die letzte der vier Funktionen ist noch nicht belegt. Sie ist freigehalten worden für spätere Erweiterungen.

Es ist noch wichtig zu wissen, welche Register nach einem Funktionsaufruf verändert sein können. Die System-Libraries verändern die Register d0, d1, a0 und a1. Dies kann natürlich bei "privaten" Libraries unterschiedlich sein.

Nachdem wir den Aufbau der Library besprochen haben, folgen nun die Funktionen, die uns Exec für die Verwaltung der Libraries zur Verfügung stellt. Einige Funktionen werden wir erst im Kapitel 9 besprechen.

AddLibrary	**=**	**-396 (Exec-Library)**

***LibraryNode a1** < Zeiger auf eine Library-Node-Struktur, die mit Hilfe der Funktion MakeLibrary erstellt worden ist.

Erklärung Die Funktion AddLibrary trägt die angegebene Library in die Library-Liste ein.

CloseLibrary	=	**-414 (Exec-Library)**

***LibraryNode a1** < Zeiger auf die Library-Node-Struktur der zu schließenden Library.

Erklärung Durch den Aufruf dieser Funktion kann die geöffnete Library wieder geschlossen werden (lib_OpenCnt wird verringert).

MakeFunctions	=	**-90 (Exec-Library)**

***Target a0** < Addresse des Bereichs, in dem die Sprungtabelle angelegt werden soll.

***Funct.Array a1** < Zeiger auf eine Tabelle, die entweder absolute 32-Bit-Adressen oder Offset Werte (16 Bit) enthalten kann. Soll eine Tabelle mit Offset-Werten benutzt werden, so muß sie mit dem Wert $FFFF (-1W) beginnen. Die Liste muß in beiden Fällen immer mit einer -1 abgeschlossen sein.

FuncDispBase a2 < Wenn die in FunctionArray übergebene Tabelle Offset-Werte enthält, muß hier die relative Adresse für die Offsets angegeben sein. Sonst sollte der Wert mit Null initialisiert werden.

TableSize d0 > Nachdem die Funktion aufgerufen wurde, erhält man die Größe der Sprungtabelle in d0 zurück.

Erklärung Durch die MakeFunctions-Routine wird eine, durch die angegebenen Parameter spezifizierte, Sprungtabelle erstellt (Näheres zu dieser Funktion siehe Kapitel 9).

MakeLibrary	=	**-84 (Exec-Library)**

***funcInit a0** < Tabelle von absoluten Adressen oder Offsets (identisch mit der Tabelle FunctionArray von MakeFunction). Die Offsetwerte würden sich hierbei immer auf die Anfangsadresse ihrer Tabelle beziehen.

***structInit a1** < Zeiger auf eine Initialisierungs-Tabelle wie sie bei der Funktion InitStruct (siehe "Sonstige Funktionen") benutzt wird. Die Initialisierungs-Tabelle dient dazu, die Library-Struktur anzulegen. Wird hier eine Null eingetragen, muß man die Library-Struktur selbständig einrichten.

*libInit	a2	<	Zeiger auf eine Routine, die nach der MakeLibrary-Funktion aufgerufen wird. Diese Routine kann die weitere Initialisierung der Library-Struktur übernehmen. Dazu bekommt sie im Register d0 einen Zeiger auf die erstellte Library-Struktur und in a0 die Adresse der Segmentliste übergeben. Nachdem die Routine beendet wurde, wird der Wert, der im Datenregister 0 zurückgegeben wurde, als lib_Node (Basisadresse) dem Programm übergeben. Will man auf eine eigene Routine verzichten, initialitisiert man den Zeiger mit Null.
dataSize	d0	<	Größe der Library-Struktur, die angelegt werden soll.
***codeSize**	d1	<	Zeiger auf Segment-Liste der Library-Datei.
***libNode**	d0	>	Zeiger auf den Library-Knoten der erstellten Library (Basisadresse oder Null).

Erklärung

Durch die Funktion MakeLibrary wird eine Library erstellt, die durch die übergebenen Parameter bestimmt wurde. Näheres finden sie im Kapitel 9 und 11.

OldOpenLibrary = -408 (Exec-Library)

***LibName**	a1	<	Zeiger auf eine Zeichenkette mit dem Namen der Library, die geöffnet werden soll.
***LibraryNode**	d0	>	Die Funktion liefert den Zeiger auf die Node-Struktur der gesuchten Library zurück. Ist ein Fehler aufgetreten, wird in d0 eine Null übergeben.

Erklärung

Die OldOpenLibray-Funktion löscht das Datenregister 0 und ruft die Funktion OpenLibrary auf. Diese Funktion durchsucht die Library-Liste nach dem angegebenen Namen und gibt die Adresse der betreffenden Node-Struktur zurück. Außerdem wird der lib_OpenCnt-Zähler erhöht. Ist die Library nicht in der Liste enthalten, wird versucht, sie von Diskette nachzuladen.

OpenLibrary	=	**-512 (Exec-Library)**

***LibName**	**a1**	<	Zeiger auf die Zeichenkette (die mit einem Null-Byte abgeschlossen werden muß) mit dem Namen der Library, die geöffnet werden soll.
Version	**d0**	<	Wird eine besondere Version der Library benötigt, kann in d0 die Mindestversionsnummer angegeben werden. Gibt es keine Version, die für die Mindestversionsnummer ausreicht, wird die Funktion abgebrochen und d0 mit Null initialisiert. Ist die Version der Library belanglos, kann eine Null übergeben werden.
***LibraryNode**	**d0**	>	Die Funktion liefert den Zeiger auf die Node-Struktur der gesuchten Library mit der angegebenen Mindestversionsnummer zurück. Ist ein Fehler aufgetreten, wird in d0 eine Null zurückgegeben.
Erklärung			Die OpenLibrary-Funktion durchsucht die Library-Liste nach dem angegebenen Namen und gibt die Adresse der betreffenden Node-Struktur zurück. Außerdem wird der lib_OpenCnt Zähler erhöht. Ist die Library nicht in der Liste enthalten, wird versucht, sie von der Diskette nachzuladen.

RemLibrary	=	**-402 (Exec-Library)**

***LibraryNode**	**a1**	<	Zeiger auf die Basisadresse bzw. Adresse der Library-Node-Struktur, die entfernt werden soll.
Error	**d0**	>	Konnte die Library nicht entfernt werden, wird in d0 der Fehlercode übergeben. Ist kein Fehler aufgetreten, erhält man eine Null.
Erklärung			Mit dieser Funktion kann man eine Library aus der Library-Liste entfernen.

SetFunction	=	**-420 (Exec-Library)**

Offset	**a0**	<	Negativer Offset der Funktion, die geändert werden soll

***LibraryNode**	**a1**	<	Basisadresse der Library bzw. Adresse des Knotens, dessen Sprungtabelle geändert werden soll
NewAddress	**d0**	<	Neue Sprungadresse, die eingetragen werden soll.
OldAddress	**d0**	>	Nach dem Aufruf erhalten wir die alte Sprungadresse der Routine zurück.
Erklärung			Die Sprungadresse wird in der Sprungtabelle der definierten Library auf die angegebene Routine gesetzt. Außerdem wird die Prüfsumme der Sprungtabelle neu berechnet. Dies ist eine ganz nützliche Funktion, um sich in das System "einzuhängen". Virus-Programme benutzen diese Möglichkeit, um aktiviert zu werden. Natürlich kann man die Sprungtabelle auch von "Hand" manipulieren. Jedoch sollte man dann nicht vergessen, die Prüfsumme mit Hilfe der SumLibrary-Funktion zu errechnen.

SumLibrary = -426 (Exec-Library)

***Library**	**a1**	<	Zeiger auf die Library-Struktur der Library, deren Checksumme neu berechnet werden soll.
Erklärung			SumLibrary berechnet die Checksumme einer Library neu.

Zum Abschluß wollen wir uns die Anwendung der Funktion SetFunction ansehen. Dazu schauen wir uns folgendes Programm, welches die Funktion AllocMem der Exec-Library durch eine eigene Routine ersetzt, an.

```
*
* Kapitel 8
* Demonstrationsprogramm für die SetFunction-Funktion
*

ExecBase    =       4
SetFunction =       -420
Forbid      =       -132
Permit      =       -138

Start:
        move.l  ExecBase,a6     ; Task-Switching sperren
        jsr     Forbid(a6)      ;
```

```
        move.l  a6,a1           ; Adresse der Library nach a1
        move.l  #-198,a0        ; Offset der zu ändernden Funktion
        move.l  #MyOpen,d0      ; Adresse der neuen Funktion
        jsr     SetFunction(a6)         ; ändern

        move.l  d0,OldJump+2    ; alte Adresse speichern

        jsr     Permit(a6)      ; Task-Switching zulassen


        ; Programm ...


        move.l  ExecBase,a6     ; Task-Switching sperren
        jsr     Forbid(a6)      ;

        move.l  a6,a1           ; alte Adresse wieder einsetzen
        move.l  #-198,a0        ;
        move.l  OldJump+2,d0    ;
        jsr     SetFunction(a6)         ;

        jsr     Permit(a6)      ; Task-Switching zulassen

        rts                     ;

MyOpen:                         ; Neue Funktion
        move.l  d0,-(a7)        ; d0 retten
        move.l  #$800,d0        ; Schleifenwert
Loop:   move.w  d0,$dff180      ; Hintergrundfarbe ändern
        sub.l   #1,d0           ; Schleife fertig?
        bne     Loop            ;
        move.l  (a7)+,d0        ; d0 restaurieren

OldJump:                        ; Damit nicht der Ablauf gestört
        jmp     MyOpen          ; wird rufen wir jetzt die
                                ; eigentliche Funktion auf
```

Programm 8.9: Demonstration SetFunction

8.6 Devices (Gerätetreiber)

Um die Kommunikation mit der Außenwelt leichter zu gestalten, stellt das System sogenannte Devices (Gerätetreiber) zur Verfügung. Diese Devices sind für einen bestimmten Bereich zuständig und enthalten alle nötigen Funktionen, um ein Gerät zu steuern.

Folgende Devices stehen uns zur Verfügung:

"audio.device"	Steuerung der Tonausgabe

"console.device"	Ein-/Ausgabe-Funktionen für Fenster
"gameport.device"	Kontrolle der GamePorts
"input.device"	Sammeldevice für Eingabeereignisse
"keyboard.device"	Umrechnung von Tastendrücken in ASCII-Codes
"narrator.device"	Sprachausgabe
"parallel.device"	Datenübertragung über den Parallel-Port
"printer.device"	Umwandlung von Texten in Drucker-Steuercodes
"serial.device"	Datenübertragung über den seriellen Port
"trackdisk.device"	Steuerung der Diskettenlaufwerke

Der Aufbau eines Devices ist zum größten Teil mit dem einer Library identisch. So wird z.B. auch eine Library-Struktur mit Sprungtabelle angelegt, die mindestens sechs Standardfunktionen enthalten muß. Diese Funktionen sind die vier Library-Standard-Funktionen, sowie die BeginIO- und die AbortIO-Funktion.

8.6.1 Sinn und Zweck der Devices

Natürlich stellt sich an dieser Stelle die Frage, warum man diese Funktionen nicht direkt mit einer Library zusammengefaßt hat. Dies liegt an folgendem Problem:

Stellen wir uns vor, ein Task versucht, einen bestimmten Block von der Diskette zu laden. Als der Schreib-/Lesekopf mit Hilfe der zugehörigen Routinen positioniert wurde, wird er durch einen Interrupt unterbrochen. Der nächste Task, mit höherer Priorität, will auch einen Block von der Diskette lesen. Dieser liegt jedoch an einer anderen Position und der Kopf wird erneut plaziert. Bekommt nun der erste Task die CPU wieder, geht er davon aus, daß der Kopf immer noch an der Stelle steht, an der er ihn zurückgelassen hat. Da jedoch der zwischengeschobene Task auch auf das Diskettenlaufwerk zugegriffen und den Schreib-/Lesekopf verschoben hat, wird der falsche Block von der Diskette geladen.

Um dieses Problem zu umgehen, wird der gesamte Zugriff auf ein Gerät von einem Device übernommen. Dieses Device richtet einen eigenen Task ein, der den Zugriff auf das entsprechende Gerät geschlossen abwickelt. Dieser Task läuft parallel zu den anderen Tasks ab.

8.6.2 Ablauf einer Device-Operation

Gehen wir von der oben besprochenen Situation aus, so können wir folgenden Ablauf beobachten: Zunächst wird das Device geöffnet und die entsprechende Nachrichten-Struktur für die Kommunikation mit dem angegebenen Device prepariert. Nun kann man die Nachricht mit den Informationen für das Device füllen und sendet sie mit der Funktion DoIO oder SendIO (wird gleich erklärt) an den Message-Port des Device-Tasks.

Sofern die Nachricht mit SendIO abgeschickt wurde, kann der Task an anderen Aufgaben, parallel zum Diskettenzugriff, weiterarbeiten. Mit der Funktion CheckIO kann er testen, ob die Nachricht schon bearbeitet worden ist.

Aus der Sicht des Device-Tasks spielt sich das wie folgt ab: Der Task für das Device erhält, wie jeder andere Task, ab und zu die CPU. In dieser Zeit kontrolliert er, ob an seinen Message-Port eine Nachricht gesendet wurde. Ist dies der Fall, holt er den Zeiger auf die Message-Struktur mittels der GetMsg-Funktion und führt die angegebene Operation aus. Ist die Aufgabe beendet, wird an den ReplyPort, der in der Message-Struktur eingetragen ist, die Nachricht (mittels ReplyMsg) zurückgeschickt. Nun ist das Device bereit, weitere Aufträge auszuführen und holt die nächste Nachricht vom Message-Port.

8.6.3 Routinen zur Arbeit mit Devices

Wie man erkennt, ist die Kommunikation mit den Devices bzw. dem zuständigen Task nicht ganz einfach. Deshalb stellt uns Exec einige Funktionen zur Verfügung, die uns die Arbeit abnehmen. Beginnen wollen wir mit der DoIO-Funktion. Sie benötigt lediglich einen Zeiger auf eine IORequest-Struktur, welche zuvor initialisiert wurde.

DoIO		=	**-456 (Exec-Library)**
***IORequest**	**a1**	<	Zeiger auf eine IORequest-Struktur, die gesendet werden soll.
Ready	**d0**	>	Nummer des Fehlers, der aufgetreten ist oder eine Null
Erklärung			Durch die DoIO-Funktion wird eine IORequest-Struktur verschickt und gewartet, bis die Nachricht bearbeitet wurde.

Die DoIO-Funktion der Exec-Library liest aus der übergebenen IORequest-Struktur die Adresse der Library-Struktur des Devices aus und ruft dann die Funktion BeginIO auf. Diese Funktion sendet die Nachrichten-Struktur an den Message-Port, des vom Device eigenhändig eingerichteten Tasks.

Nachdem die BeginIO-Funktion aufgerufen wurde, wird die WaitIO-Funktion bearbeitet, die eine Unterfunktion von DoIO ist.

WaitIO	=	**-474 (Exec-Library)**

***IORequest** **a1** < Zeiger auf eine IORequest-Struktur.
Error **d0** > In d0 findet man entweder den Fehlercode oder eine Null.

Erklärung Durch die Funktion WaitIO wird der Task solange in die TaskWait-Liste aufgenommen, bis die angegebenen Nachricht bestätigt worden ist.

Durch sie wird getestet, ob die Nachrichten-Struktur (IORequest) schon vom Device-Task zurückgeschickt worden ist. Hierbei wird lediglich in der Node-Struktur, der IORequest-Struktur, nachgesehen ob der Typ der Daten mit 7, also nt_ReplyMsg, initialisiert wurde. Ist das nicht der Fall, wird der Task durch die Wait-Funktion "schlafen geschickt". Dabei werden die SignalBits aus der ReplyPort-Struktur benutzt. Kommt ein Signal an, wird die IORequest-Struktur wiederum überprüft.

Sollte die IORequest-Struktur wirklich bestätigt worden sein, so wird sie noch aus der MsgList des ReplyPorts entnommen und der Fehlercode aus der IORequest-Struktur in das Datenregister d0 geschoben. Damit ist sowohl die WaitIO- als auch die DoIO-Funktion beendet.

Wie wir bei DoIO gesehen haben, wird der Task angehalten und mittels der Wait-Funktion "schlafen gelegt". Für ein Multitasking-System ist das jedoch nicht befriedigend. Wenn schon ein unabhängiger Task für die Devices zuständig ist, müßte es doch auch möglich sein, den eigenen Task weiterlaufen zu lassen. Genau diese Möglichkeit eröffnet uns die SendIO-Funktion. Durch sie können wir unsere Auf-forderung senden und dennoch weiterarbeiten.

SendIO	=	**-462 (Exec-Library)**

***IORequest** **a1** < Zeiger auf IORequest-Struktur.

Erklärung Durch die Funktion SendIO wird lediglich die Grundfunktion BeginIO (-$30) aufgerufen, die die IORequest-Struktur an den Device-Task weiterleitet. Danach kann der Task an seinen Aufgaben weiterarbeiten. Wenn die Operation des Device erledigt ist, erhält der Task über den angegebenen ReplyPort seine Nachricht zurück.

Die SendIO-Funktion gleicht im Grunde der DoIO-Funktion genau. Jedoch wird, nachdem die Funktion BeginIO der Library angesprungen worden ist, nicht auf die Bestätigung der Nachricht gewartet. Das Programm kann jetzt an anderen Aufgaben weiterarbeiten. Hat sie diese erledigt, so können wir die Unterfunktion von DoIO (WaitIO) benutzen um zu warten, bis unsere Nachricht bearbeitet wurde.

Es gibt jedoch noch eine zweite Möglichkeit mit der man testen kann, ob die Nachricht schon bearbeitet wurde. Diese Funktion heißt CheckIO und kontrolliert lediglich, ob die angegebene Nachricht bestätigt worden ist. Wenn das nicht der Fall ist, kann man weiterarbeiten und überprüft die Nachricht nochmals zu einem späteren Zeitpunkt.

CheckIO	=	**-468 (Exec-Library)**

***IORequest**	a1	<	Zeiger auf IORequest-Struktur, auf die gewartet werden soll
Ready	d0	>	Ist die angegebene Nachricht fertig, erhält man eine Null zurück.
Erklärung			Testet, ob die angegebene IORequest-Struktur schon an ihren ReplyPort zurückgeschickt worden ist.

Wie bei WaitIO wird auch bei CheckIO nur überprüft, ob der Node-Typ der Nachricht schon als nt_ReplyMsg gekennzeichnet wurde. Allerdings erhält man durch CheckIO nur die Information, ob die Nachricht bestätigt worden ist. Anschließend muß sie, mittels der GetMsg-Funktion, aus der Msg-Liste des ReplyPorts entfernt werden.

Als letzte IO-Funktion sollte noch AbortIO erwähnt werden. Sie bricht den angegebenen IO-Prozeß ab. Dabei wird die gleichnamige Funktion AbortIO des Devices aufgerufen.

AbortIO	=	**-480 (Exec-Library)**

***IORequest**	a1	<	Zeiger auf IORequest-Struktur.
Erklärung			Durch die Funktion AbortIO kann ein IO-Request abgebrochen werden.

Sicher ist dieser Teil etwas kompliziert, da man leicht die Namen der Exec-Funktionen und der Device-(Library)-Funktionen vewechseln kann. Die folgende Tabelle soll Ihnen einen Überblick geben.

Exec-Funktionen:

OpenDevice	Öffnet das angegebene Device mit den übergebenen Parametern. Außerdem wird die IORequest-Struktur eingerichtet. Das heißt, daß die Einträge io_Unit gelöscht werden, und io_Device mit dem Zeiger auf die Library-Struktur des Device belegt wird. Dann wird die Open-Funktion des Devices (-6) aufgerufen, welche private Installationen vornehmen kann. Der Device-Task und der zugehörige Message-Port sind zu diesem Zeitpunkt schon eingerichtet worden. Dies erledigt die Init-Routine der Resident-Struktur (diese Details werden im Kapitel 11 näher erklärt). Sollte ein Fehler aufgetreten sein, wird der Eintrag io_Error und das Datenregister d0 mit -1 belegt.
CloseDevice	Die Funktion CloseDevice (Exec) führt lediglich die device-spezifische Close-Funktion (-12) aus.
RemDevice	Bei dieser Funktion wird nur die Library-Struktur des Devices aus der Device-Liste von Exec entnommen.
DoIO	Setzt den Eintrag io_Flags der angegebenen Struktur auf 1 (Quick-Bit gesetzt) und führt die device-spezifische Funktion BeginIO aus. Danach wird die WaitIO-Funktion abgearbeitet.
SendIO	Löscht den Eintrag io_Flags, führt die BeginIO-Routine aus und kehrt zum aufrufenden Task zurück.
WaitIO	Wartet solange, bis die abgeschickte Nachricht bearbeitet wurde. Danach wird die Nachrichten-Struktur aus der Liste der Nachrichten des Reply-Ports entnommen.
CheckIO	Testet, ob die angegebene Nachricht bearbeitet worden ist. Ist dies der Fall, so erhält man in d0 den Zeiger auf die Nachricht-Struktur. Diese ist jedoch noch nicht aus der Liste des Reply-Ports entnommen worden!
AbortIO	Ruft die gleichnamige device-spezifische Funktion AbortIO auf.

Auch der Aufbau im Speicher soll noch einmal explizit an einer Skizze verdeutlicht werden:

Library-Struktur, die vom Device angelegt worden ist. Dazu gehört auch eine Sprungtabelle, die mindestens die sechs Standardfunktionen aufnehmen muß!

BeginIO	**AbortIO**	**Standardfunktionen** für Libraries
Sendet die Nachricht an den MessagePort des DeviceTasks	Bricht die angegebene Operation ab	Open/Close Expunge/ExtFunc

Sendet die Nachrichten an den MessagePort des Tasks, der vom Device erstellt worden ist.

MessagePort des Device-Tasks. Er enthält alle Anträge der erwünschten Operationen des Gerätes.

Device-Task. Er ist für alle Operationen des Gerätes verantwortlich. Er kontrolliert seinen MessagePort und erledigt die dort eingegangenen Aufträge nach dem fifo-Prinzip. Wurde eine Operation ausgeführt, wird die Nachricht an den, in der IOReq.-Struktur angegebenen, ReplyPort zurückgesendet.

Standardfunktionen eines Devices:	**Sonderfunktionen eines Devices:**
CmdInvalid/CmdReset/ CmdRead/CmdWrite/CmdUpdate CmdClear/CmdStop/CmdStart/ CmdFlush	CmdNonStd + n

(Außerdem gibt es auch Devices, die nicht nur einen Task anlegen. So erstellt das Trackdisk-Device für jedes Laufwerk einen eigenen Task, der für die Steuerung des Laufwerks verantwortlich ist.)

8.6.4 IORequest- und IOStdRequest-Struktur

Jetzt fehlen uns noch zwei Strukturen, die für die erweiterte Nachrichtenübermittlung verwendet werden. Sie heißen IORequest- und IOStdRequest, was soviel bedeutet wie Eingabe/Ausgabe-(Standard-)Anfrage-Struktur. Beginnen wollen wir mit der IORequest-Struktur:

IORequest-Struktur:

```
00    dc.l   *io_Succ        ;┐          ┐
04    dc.l   *io_Pred        ;│          │
08    dc.b    io_Type        ;│ Node     │
09    dc.b    io_Pri         ;│          │
10    dc.l   *io_Name        ;┘          │ Message
                                         │
14    dc.l   *io_ReplyPort   ;           │
18    dc.w    io_Length      ;           ┘

20    dc.l   *io_Device      ; Zeiger auf Device-Struktur
24    dc.l   *io_Unit        ; Zeiger auf Unit-Struktur
28    dc.w    io_Command     ; IO-Kommando
30    dc.b    io_Flags       ; Device-Flags
31    dc.b    io_Error       ; Fehler bei IO-Ausführung
32            io_SIZEOF
```

***io_Succ - io_Length**
Wie man sieht, setzt sich die IORequest-Struktur aus einer Message-Struktur und einigen Spezialeinträgen zusammen. Die Zusammensetzung der Message-Struktur können sie aus dem Kapitel für Messages und Ports entnehmen.

***io_Device**
Dieser Eintrag enthält die Adresse eines Device-Knotens, die durch die Funktion OpenDevice eingetragen wurde. Hierbei handelt es sich um die Adresse der Library-Struktur, die für das Device angelegt wurde und über die die sechs Standardfunktionen angesprungen werden können.

***io_Unit**
Der Eintrag io_Unit kann einen Zeiger auf eine Unit-Struktur enthalten, in der für den Device-Task wichtige Daten abgelegt sind. Dieser Wert muß allerdings von der devicespezifischen Open-Funktion gesetzt werden.

io_Command
In io_Command muß der Befehl abgelegt werden, welcher ausgeführt werden soll. Dabei stehen einige Standardbefehle und auch device-abhängige Spezialbefehle zur Verfügung.

IO-Kommando	Wert	Bedeutung
cmd_Invalid	00	ungültiger Befehl
cmd_Reset	01	Device zurücksetzen
cmd_Read	02	Daten in Puffer lesen
cmd_Write	03	Daten in Puffer schreiben
cmd_UpDate	04	Daten aus Puffer an Gerät schicken
cmd_Clear	05	Puffer löschen
cmd_Stop	06	Arbeit des Devices sperren
cmd_Start	07	Arbeit des Devices erlauben
cmd_Flush	08	alle IORequest-Strukturen streichen
cmd_NonStd	09	Sonderkommando

cmd_Invalid	Ungültiger Befehl !
cmd_Reset	Durch den cmd_Reset Befehl wird das Device zurückgesetzt und die Liste der IORequest-Strukturen geleert.
cmd_Read	Wie der Name schon sagt, soll das Device Daten lesen.
cmd_Write	Auch der folgende Befehl dürfte sich selbst erklären. Er veranlaßt das Device, Daten zu schreiben.
cmd_UpDate	Mittels der cmd_UpDate Anweisung wird das System aktiviert, die Daten aus dem Puffer dem Gerät zu übergeben. Das kann wichtig sein, wenn man z.B. auf eine Diskette einen Block schreiben will. Man benutzt zunächst den cmd_Write Befehl, der die angegebenen Daten in den Puffer übernimmt. Erst durch cmd_UpDate werden die Daten aus dem Puffer auf die Diskette geschrieben.
cmd_Clear	Der Zwischenspeicher des Devices wird gelöscht. Achtung: Man sollte mit dem Befehl cmd_UpDate sicherstellen, daß die Daten wirklich übertragen worden sind bevor man den Puffer löscht.
cmd_Stop	Der cmd_Stop Befehl erlaubt es, ein Device lahmzulegen. Dabei wird nur die Abarbeitung der IO-Request-Strukturen gestoppt. Neue Strukturen dieser Art können immer noch in die Liste aufgenommen werden.
cmd_Start	Wurde die Arbeit eines Devices mittels des cmd_Stop Befehls angehalten, kann man es mit dem Kommando cmd_Start wieder in Gang setzen.
cmd_Flush	Durch cmd_Flush wird die Liste der IORequest-Strukturen des Devices geleert. Das heißt, alle Meldungen werden zurückgesandt und der entsprechende Fehlercode übermittelt.
cmd_NonStd	Ab dieser Kennung fangen Sonderkommandos an

io_Flags
Durch die, im Eintrag io_Flags, gesetzten Werte wird die Bearbeitung der IORequest-Struktur beeinflußt. So ist hier z.B. das Quick-Flag enthalten, welches von der DoIO-Funktion getestet wird. Näheres dazu finden sie im Kapitel 12.

io_Error
Ist ein Fehler aufgetreten, findet man an dieser Stelle den Fehlercode. Er wird auch im Datenregister d0 nach den Funktionen DoIO und WaitIO übergeben. Die Bedeutung der Fehlercodes ist abhängig vom Device.

Die zweite Struktur, die wir uns ansehen wollten, heißt IOStdRequest (Eingabe/Ausgabe-Standard-Anfrage). Sie enthält neben einer "normalen" IORequest-Struktur noch weitere Einträge.

IOStdRequest-Struktur:

```
00      dc.l    *io_Succc          ;┐      ┐         ┐
04      dc.l    *io_Pred           ;│      │         │
08      dc.b     io_Type           ;│ Node │         │
09      dc.b     io_Pri            ;│      │         │
10      dc.l    *io_Name           ;┘      │ Message │
                                           │         │
14      dc.l    *io_ReplyPort      ;       │         │ IORequest
18      dc.w     io_Length         ;       ┘         │
                                                     │
20      dc.l    *io_Device         ;                 │
24      dc.l    *io_Unit           ;                 │
28      dc.w     io_Command        ;                 │
30      dc.b     io_Flags          ;                 │
31      dc.b     io_Error          ;                 ┘

32      dc.l     io_Actual         ; Anzahl der übertragenen Bytes
36      dc.l     io_Length         ; Anzahl der zu übertr. Bytes
40      dc.l     io_Data           ; Zeiger auf Daten
44      dc.l     io_Offset         ; Offsetwert für Device
48               io_SIZEOF
```

io_Actual
Nachdem der angegebene Befehl ausgeführt worden ist, enthält der Eintrag io_Actual die Anzahl der übertragenen Bytes, oder auch andere Rückgabewerte (device-abhängig).

io_Length
Durch den Wert io_Lenght wird die Anzahl der Bytes angegeben, die übertragen werden sollen.

io_Data
In io_Data muß die Adresse, ab der die Daten-Übertragung erfolgen soll, eingetragen werden.

io_Offset
Hier kann ein Offset-Wert angegeben werden, der z.B. beim Trackdisk-Device (TDD) den Block angibt, ab dem auf die Diskette zugegriffen werden soll.

Einige Devicetypen haben spezielle Nachrichten-Strukturen, die meist auf den beiden vorangegangenen Strukturen aufbauen.

Bevor wir uns das Demonstrationsprogramm ansehen, wollen wir uns nochmal überlegen, welche Schritte notwendig sind, um ein Device zu benutzen.

1. Zunächst müssen wir einen ReplyPort (ein ganz gewöhnlicher MessagePort) erstellen, den wir mit AddPort in die System-Liste der Public-Ports aufnehmen können (nicht müssen!).

2. Dann initialisieren wir unsere IORequest- oder IOStdReq-Struktur (oder was das Device erwartet).

3. Anschließend öffnen wir das Device, wobei wir neben einigen anderen Parametern einen Zeiger auf unsere Nachrichtenstruktur übergeben.

4. Jetzt können wir durch DoIO und SendIO unsere Nachricht verschicken und sie mittels WaitIO, CheckIO und AbortIO kontrollieren bzw. abbrechen.

5. Um die Kommunikation mit dem Device abzubrechen, muß man lediglich das Device schließen und dann den Port mit der RemPort-Funktion entfernen.

Jetzt können wir uns an das Demonstrationsprogramm wagen. Es ist <u>nicht</u> lauffähig und soll nur das Prinzip veranschaulichen.

```
*
* Kapitel 8
* Demonstrationsprogramm für die Anwendung der Device-Funktionen
*

ExecBase      =       4
FindPort      =       -390
AllocMem      =       -198
FreeMem       =       -210
AddPort       =       -354
RemPort       =       -360
AllocSignal   =       -330
FreeSignal    =       -336
DoIO          =       -456
OpenDevice    =       -444
CloseDevice   =       -450


Start:  move.l  ExecBase,a6
        sub.l   a1,a1           ; Name ist belanglos
        move.l  #0,d0           ; Priorität = 0
        bsr     CreatePort      ; Port anlegen
        move.l  d0,io_ReplyPort        ; Port in IORequest-Struktur
        beq     PortError       ; eintragen

        lea     DeviceName,a0   ; Name des Devices
        lea     IORequest,a1    ; Zeiger auf IORequest-Struktur
        moveq   #2,d0           ; Unit
        moveq   #0,d1           ; Flags
        jsr     OpenDevice(a6)  ; Device öffnen
        tst.l   d0              ; Fehler aufgetreten?
        bne     DevError        ; Dann verzweigen
```

```
        ; ...

        move.w   #2,io_Command    ; io_Command = CMD_Read
        move.l   #Mem,io_Data     ; io_Data = Mem
        move.l   #0,io_Offset     ; io_Offset = 0
        move.l   #512,io_Length   ; io_Length = 512
        lea      IORequest,a1     ; IORequest-Struktur
        jsr      DoIO(a6)         ; IORequest ausführen

        ; ...

        lea      IORequest,a0     ; IORequest-Struktur
        jsr      CloseDevice(a6)        ; Device schließen

DevError:
        move.l   io_ReplyPort,a0        ; Zeiger auf Port
        bsr      DeletePort       ; Port löschen

PortError:
        rts                       ; Programm beenden

* Datenbereich


DeviceName:   dc.b    "xyz.device",0       ; Name des Devices
              even

IORequest:    dc.l    0,0                  ; io_Succ, io_Pred
              dc.b    0,0                  ; io_Type, io_Pri
              dc.l    0                    ; io_Name
io_ReplyPort: dc.l    0                    ; io_ReplyPort
              dc.w    IOEnd-IORequest      ; io_Length
              dc.l    0                    ; io_Device
              dc.l    0                    ; io_Unit
io_Command:   dc.w    0                    ; io_Command
              dc.b    0                    ; io_Flags
              dc.b    0                    ; io_Error
io_Actual:    dc.l    0                    ; io_Actual
io_Length:    dc.l    0                    ; io_Length
io_Data:      dc.l    0                    ; io_Data
io_Offset:    dc.l    0                    ; io_Offset

IOEnd:

Mem:          ds.b    1000                 ; Datenbereich
```

Bild 8.10: Benutzung der Device-Routinen

8.6.5 Sonstige Device-Routinen

Bevor wir dieses Kapitel beenden, wollen wir uns noch die weiteren Device-Funktionen der Exec-Library ansehen.

AddDevice = -432 (Exec-Library)

***Device**	**a1**	<	Zeiger auf die Device-Struktur (eigentlich eine Library-Struktur), die in die Liste aufgenommen werden soll.
Erklärung			AddDevice nimmt ein neues Device in die Liste aller Device-Strukturen auf.

CloseDevice = -450 (Exec-Library)

***IORequest**	**a1**	<	Zeiger auf eine IORequest-Struktur.
Erklärung			Durch den Aufruf von CloseDevice wird dem Device mitgeteilt, daß der Task das Device nicht mehr benötigt. Wie bei den Libraries kann ein Device aus dem Speicher "geworfen" werden.

OpenDevice = -444 (Exec-Library)

***DevName**	**a0**	<	Zeiger auf den Namen des Devices, welches geöffnet werden soll.
***IOReques**	**a1**	<	Zeiger auf eine initialisierte IORequest-Struktur.
Unit	**d0**	<	Nummer der Einheit. Bei der Benutzung des trackdisk.device kann man hier z.B. die Laufwerkseinheit (0-3) angeben.
Flags	**d1**	<	Flags (wird nicht von allen Devices benötigt).
Error	**d0**	>	Nachdem die Funktion aufgerufen wurde, enthält d0 entweder eine Null oder die Nummer des aufgetretenen Fehlers.
Erklärung			Durch den Aufruf der OpenDevice-Funktion wird das angegebene Device geöffnet und die angegebene IORequest-Struktur initialisiert. Dabei werden die Werte für io_Unit und io_Device eingetragen.

RemDevice = -438 (Exec-Library)

***Device**	**a1**	<	Zeiger auf ein Device.
Error	**d0**	>	Null oder Nummer des Fehlers.
Erklärung			Durch RemDevice wird ein Device aus der Device-Liste von Exec entfernt.

8.7 Resources

Die Resources bilden eine noch tiefere Schicht der Programmierung und werden z.B. von den Devices benutzt. Ihr Aufbau und ihre Anwendung ist mit den Libraries vergleichbar. Da die Resources jedoch zu speziell sind, wollen wir nicht auf sie eingehen.

8.8 Externe Prozessor-Ausnahmen (Interrupts)

Wie wir schon im Kapitel über interne Prozessor-Ausnahmen besprochen haben, gibt es auch Exceptions, die von externen Bausteinen ausgelöst werden können. Diese Interrupts ermöglichen es, schnell auf Signale von außen zu reagieren.

Auch als Programmierer kann man von dieser Einrichtung profitieren. Man benötigt lediglich eine initialisierte Interrupt-Struktur.

Interrupt-Struktur:

```
00      dc.l    *is_Succ           ;┐
04      dc.l    *is_Pred           ;│
08      dc.b     is_Type           ;│ Node-Struktur
09      dc.b     is_Pri            ;│
10      dc.l    *is_Name           ;┘

14      dc.l    *is_Data           ; Zeiger auf Datenteil
18      dc.l    *is_Code           ; Zeiger auf Interrupt-Routine
22               is_SIZEOF
```

***is_Succ, *is_Pred, is_Type, is_Pri, *is_Name**
Die ersten fünf Einträge dienen dazu, die Interrupt-Strukturen in eine Liste aufzunehmen.

***is_Data**
Zeiger auf den Datenbereich, der vom Interrupt-Programm benutzt werden soll.

***is_Code**
Zeiger auf die Interruptroutine, die ausgeführt werden soll.

Hat man die Interrupt-Struktur initialisiert, kann man sie mittels der Funktion SetIntVector einem angegebenen Interrupt zuordnen.

SetIntVector	=	-162 (Exec-Library)

Interrupt	a1	<	Zeiger auf die Interrupt-Struktur, die neu eingesetzt werden soll.
IntNum	d0	<	Nummer des Interrrupts, für den die Interrupt-Struktur gelten soll (siehe unten).
Interrupt	d0	>	Nachdem man die Funktion aufgerufen hat, erhält man in d0 die Adresse der alten Interrupt-Struktur.
Erklärung			Durch die Funktion SetIntVector kann eine neue Interrupt-Struktur für den angegebenen Interrupt verwendet werden. Dabei wird die IntVector-Struktur des angegebenen Interrupts initialisiert.

Durch die Funktion SetIntVector wird eine weitere Struktur definiert, die sogenannte IntVector-Struktur. Eine solche Struktur, die übrigens im Exec-Datenbereich liegt, gibt es für jeden Interrupt des Amiga. Sie ist wie folgt aufgebaut:

IntVector-Struktur:

```
00    dc.l    iv_Data          ; Zeiger auf Interrupt Daten
04    dc.l    iv_Code          ; Zeiger auf Programm für Interrupt
08    dc.l    iv_Node          ; Zeiger auf Interrupt-Struktur
12            iv_SIZEOF
```

iv_Data
Zeiger auf den Datenbereich für die Interrupt-Routine. Dieser Wert ist mit der in der Interrupt-Struktur angegebenen Adresse identisch.

iv_Code
Zeiger auf die Interrupt-Routine, die ausgeführt werden soll.

iv_Node
Zeiger auf die Interrupt-Struktur.

Nachdem man die Funktion SetIntVector aufgerufen hat und das IPL-Signal des entsprechenden Interrupts anliegt, wird die angegebene Routine ausgeführt.

Dabei schaltet sich eine Exec-Routine dazwischen, welche die Registerinhalte verändert. Zunächst werden die Inhalte der Register d0-1, a0-1 und a5-6 auf den Stack abgelegt. Zusätzlich befindet sich noch die Rücksprungadresse auf dem Stack. Dadurch wird es möglich, das eigene Interrupt-Programm mit einer RTS-Anweisung zu verlassen.

d1	Bitkombination, die die erlaubten Inter. anzeigt
a1	Zeiger auf den Anfang der Customchip-Register
a5	Adresse des Interrupt-Programms
a6	Zeiger auf die ExecBase-Struktur

Diese Art eines Interrupts nennt man Handler-Interrupt. Sie erlaubt lediglich, daß eine Interrupt-Routine ausgeführt wird. Bei einem Multitasking-System ist es jedoch sinnvoll, nicht nur eine Routine auszuführen, sondern mehrere hintereinander. Natürlich kann man auch einen solchen Interrupt-Typ beim Amiga erstellen: den Server-Interrupt.

Der Unterschied zu einem Handler besteht darin, daß die Einträge der IntVector-Struktur andere Bedeutungen haben.

iv_Data
Ist ein Zeiger auf eine ServerList-Struktur, welche die Interrupt-Strukturen, die ausgeführt werden sollen, enthält.

iv_Code
Zeiger auf eine Routine, die die Interrupt-Strukturen nacheinander aufruft.

iv_Node
Der letzte Eintrag hat keine Bedeutung.

Wie man sieht, unterscheiden sich die beiden Interrupt-Struktur-Typen nur wenig. Man kann sagen, daß es sich bei einem Server-Interrupt nur um einen Handler-Interrupt handelt, der eine Reihe von weiteren Interrupts verwaltet. So könnte man z.B. einen Server-Interrupt mittels der Funktion SetIntVector durch einen "normalen" Handler-Interrupt austauschen. Die Folge wäre, daß nur noch eine Routine ausgeführt würde.

Als letzte Struktur fehlt jetzt nur noch die ServerList-Struktur, mit der die Interrupts verwaltet werden können.

ServerList-Struktur:

```
00    dc.l   *sl_Head        ;┐
04    dc.l   *sl_Tail        ;│
08    dc.l   *sl_TailPred    ;│ List-Struktur
12    dc.b    sl_Type        ;│
13    dc.b    sl_Pad         ;┘

14    dc.w    sl_IntClr1     ; "Interrupt-Clear"-Wert
16    dc.w    sl_IntSet      ; "Interrupt-Set"-Wert
18    dc.w    sl_IntClr2     ; "Interrupt-Clear"-Wert
20    dc.w    sl_Pad         ; Füllwort
```

***sl_Head, *sl_Tail, *sl_TailPred, sl_Type, sl_Pad**
List-Struktur in der die einzelnen Interrupt-Strukturen verwaltet sind.

sl_IntClr1
Der Eintrag sl_IntClr1 (und auch 2) enthält einen Wort-Wert, der, wenn er in das Interrupt-Register geschrieben wird, den für diesen Server zuständigen Interrupt sperrt.

sl_IntSet
Der Wert dieses Eintrags entspricht dem Wert des Eintrags sl_IntClr1. Der Unterschied liegt lediglich darin, daß das 15. Bit gesetzt ist. Dadurch wird der gesperrte Interrupt wieder freigegeben.

sl_IntClr2
(siehe sl_IntClr1)

sl_Pad
Füllbyte, welches zur Begradigung der Struktur dient.

Bei den durch Exec installierten Server-Interrupts werden die Register d0, a1 und a5 mit Werten belegt, die für das Interrupt-Programm interessant sein könnten. Außerdem wird auf den Rückgabewert geachtet. Sollte der letzte Interrupt keinen Null-Wert zurückgeben, wird kein weiterer Interrupt ausgeführt.

d0	Zeiger auf nächsten Interrupt
a1	Zeiger auf Datenbereich des Interrupts
a5	Adresse des Interrupt-Programms

Für die **Coper-**, **VertB-**, **Blit-**, **Exter-**, **Ports-** und **NMI-Interrupts** sind die IntVector-Strukturen schon als **Server** eingerichtet. Dabei gehören die Routinen (Server-Interrupts) zur Verwaltung der Server-Listen zu Exec.

Um nun einen Interrupt in eine Server-Liste aufzunehmen, benötigt man wiederum eine Interrupt-Struktur. Sie wird dann mit Hilfe der Funktion AddIntServer in die Server-Liste aufgenommen.

AddIntServer	=	**-168 (Exec-Library)**

Interrupt	**a1**	<	Zeiger auf eine initialisierte Interrupt-Struktur, die an die Server-Liste des angegebenen Interrupt-Servers gehängt werden soll.
IntNum	**d0**	<	Nummer des Interrupts, an dessen Interrupt-Server-Liste der Interrupt angehängt werden soll.

Erklärung Durch die Funktion AddIntServer wird eine initialisierte Interrupt-Struktur an die Server-Liste der angegebenen Interrupts gehängt.

Natürlich kann man auch einen eingebundenen Interrupt wieder aus der Liste entfernen. Dazu bietet sich die Funktion **RemoveInterruptServer** an.

RemIntServer	=	-174 (Exec-Library)

Interrupt	a1	<	Zeiger auf die Interrupt-Struktur, die entfernt werden soll.
IntNum	d0	<	Nummer des Interrupts.

Erklärung Mit Hilfe der Funktion RemIntServer kann man eine durch den Befehl AddIntServer angehängte Interrupt-Struktur aus einer Interrrupt-Server-Liste entfernen.

8.8.1 Wie entstehen eigentlich Interrupts?

Nun haben wir uns angesehen, wie die Interrupt-Verwaltung vom System bewältigt wird. Doch was ist ein Interrupt und wie entsteht er eigentlich?

Ein Interrupt wird durch ein Signal eines externen Bausteins ausgelöst. Bevor jedoch dieses Signal an den Prozessor weitergegeben wird, schaltet sich Paula dazwischen.

Paula ist ein Customchip, der Ein- und Ausgabe-Operationen übernimmt, wie z.B. die Tonausgabe, sowie die Koordinierung der Daten für das Diskettenlaufwerk. Außerdem übernimmt Paula die Verwaltung der Interrupts.

Erhält Paula von einer der 14 Interruptquellen ein Signal, so wird kontrolliert, ob der entsprechende Interrupt zugelassen ist. Dies geschieht mit Hilfe des Customchip-Registers INTENA (Interrupt Enable) dessen Bits je eine Interrupt-Quelle repräsentieren.

Geht man davon aus, daß der Interrupt erlaubt war, wird das kodierte Signal an die IPL0-IPL2 (Interrupt-Pending-Level) Eingänge des Prozessors angelegt. Dabei können mehrere Interruptquellen das gleiche IPL-Signal auslösen.

Das anliegende IPL-Signal unterbricht den Prozessor bei der Arbeit. Er vergleicht das anliegende Signal mit der Signalmaske des Statusregisters. Stellt sich heraus, daß ein Interrupt höherer Priorität anliegt, wird die Maske des Statusregisters geändert, der neue Signal-Wert mit vier multipliziert und zu der Basisadresse der Interrupt-Autovektoren

addiert. An der so errechneten Position befindet sich die Adresse der Behandlungsroutine, die ausgeführt werden soll.

Wie schon erwähnt, benutzen einige Interrupts die gleichen IPL-Signale für die CPU. Dies ergibt sich durch die drei IPL-Leitungen, die lediglich 2^3 Prioritätsstufen für 14 verschiedenen Interruptquellen erlauben.

Für jede dieser IPL-Interrupts ist eine Speicherstelle festgelegt ($64-$7C), an der die Adresse der auszuführenden Behandlungsroutine vermerkt ist. Diese Routinen werden von Exec gestellt und kontrollieren zunächst, welcher Interrupt das IPL-Signal ausgelöst hat. Natürlich kann der Zeiger auf die Behandlungsroutine der einzelnen Interrupts auf eine eigene Routine umgebogen werden. Dann würde man jedoch das System aus der Interrupt-Kette ausklinken, was sich in manchen Fällen ungünstig auswirken könnte.

Die folgende Tabelle zeigt die Interruptquellen mit ihren Namen, Exec-Prioritäten, CPU/IPL-Prioritäten, Vektoradressen, Offsetwerten für den Exec-Datenbereich der Exec-IntVect-Strukturen und ihre Bedeutungen.

Name	Exec	IPL	Vektor	ExecInt	Bedeutung
SoftInt	00	1	$64	$054	Software Unterbrechung
DskBlk	01	1	$64	$060	DiskBlock übertragen
TBE	02	1	$64	$06C	serieller Port-Puffer leer
Ports	03	2	$68	$078	CIA-A/Expansion Port Int
Coper	04	3	$6C	$084	Copper-Interrupt
VertB	05	3	$6C	$090	Seitenaufbau fertig
Blit	06	3	$6C	$09C	Blitter-Interrupt
Aud2	07	4	$70	$0C0	Audio-Interrupt 0
Aud0	08	4	$70	$0A8	Audio-Interrupt 1
Aud3	09	4	$70	$0CC	Audio-Interrupt 2
Aud1	10	4	$70	$0B4	Audio-Interrupt 3
RBF	11	5	$74	$0D8	serieller Port-Puffer voll
DskSync	12	5	$74	$0E4	Syncmarkierung gelesen
Exter	13	6	$78	$0F0	CIA-B/Expansion Port Int
Inten	14	6	$78	$0FC	Copper
NMI	15	7	$7C	$108	nicht maskierbarer Interrupt

Stellen wir uns vor, Paula bekommt ein VertB-Signal (Vertical Blanking Interrupt tritt immer dann auf, wenn die letzte Bildschirmzeile vom Rasterstrahl durchlaufen wurde). Zunächst wird überprüft, ob der Interrupt im INTENA-Register zugelassen ist. Dann werden die IPL-Eingänge mit dem Prioritätswert 3 belegt. Dadurch wird die Arbeit des Prozessors unterbrochen, und das anliegende Signal mit der Signalmaske im SR (Statusregister) verglichen. Sollte im SR eine niedriegere Priorität verzeichnet sein, wird die Maske neu gesetzt und die Position der Behandlungsroutine ($6C) errechnet. Diese Behandlungsroutine ist von Exec (zumindest sollte sie von Exec sein) und überprüft, welches Ereignis (Copper, VertB oder Blit) den Interrupt ausgelöst hat. Bei

unserem Beispiel stellt die Routine fest, daß ein VertB-Interrupt anliegt. An dieser Stelle kommen wieder die IntVector-Strukturen zur Geltung, die für jeden Interrupt angelegt wurden. Aus ihnen liest nämlich die Behandlungs-routine die Adresse der Interrupt-Routine, sowie die Adresse des Datenspeicher aus und ruft die Routine auf.

Handelt es sich bei der Routine um einen Interrupt-Server, so ruft er nacheinander alle durch die im Datenbereich liegende List-Struktur verketteten Interrupts auf.

8.8.2 Zusätzliche Interrupt-Funktionen

Zum Schluß wollen wir einen Blick auf die Disable- und Enable-Funktionen werfen, die es erlauben, die Interrupts zu sperren bzw. freizugeben. Beide Funktionen benutzen die sogenannten Customchip-Register.

Bei Customchip-Registern handelt es sich um Register, die benutzt werden, um die Customchips zu steuern. Mit ihnen können z.B. Einstellungen für Auflösung, Farben, Grafikbereich, sowie für Ein- und Ausgabe gewählt werden. Sie beginnen ab der Adresse $dff000.

Für uns gibt es dort vier recht interessante Register.

$dff01c	INTENAR	Interrupt-**Ena**ble read Register
$dff09a	INTENA	Interrupt-**Ena**ble write Register
$dff01e	INTREQR	Interrupt-**Req**uest read Register
$dff09c	INTREQ	Interrupt-**Req**uest write Register

Die beiden INTENA(R)-Register enthalten die Bitmaske der Interrupts, die erlaubt sind. Die INTREQ(R)-Register enthalten die Bitmaske des gerade anliegenden Interrupts. Genau diese beiden Register werden auch von Agnus benutzt, um zu testen, ob der empfangene Interrupt vom System zugelassen war. Alle vier Register haben den gleichen Aufbau:

Name	Bit	Bedeutung
SET/CLR	15	Durch das SET/CLR-Bit wird angegeben, wie auf das Register zugegriffen werden soll: 0=löschen/1=setzen
INTEN	14	Dieses Bit stellt eine Art Hauptschalter dar. Nur wenn es gesetzt ist, können Interrupts bearbeitet werden.
EXTER	13	Externer Interrupt (CIA-B-Baustein)
DSKSYN	12	Disk-Synchronisationsmarkierung wurde gefunden
RBF	11	Eingabepuffer voll (serielle Schnittstelle)
AUD3	10	Sounddaten über Kanal 3 vollständig ausgegeben
AUD2	09	Sounddaten über Kanal 2 vollständig ausgegeben
AUD1	08	Sounddaten über Kanal 1 vollständig ausgegeben
AUD0	07	Sounddaten über Kanal 0 vollständig ausgegeben
BLIT	06	Blitter fertig
VERTB	05	Beginn der vertikalen Austastlücke erreicht
COPER	04	Interrupt durch Copper

PORTS	03	Interrupt von CIA-A-Baustein
SOFT	02	Softinterrupts
DSKBLK	01	Datenübertragung über Disk-DMA beendet
TBE	00	Ausgabepuffer voll (serielle Schnittstelle)

Wie man sieht kann man jeden Interrupt einzeln oder auch alle gleichzeitig sperren bzw. zulassen.

Als Programmierer kann man von den transparenten Registern profitieren. So hat man die Möglichkeit auszuwählen, welche Interrupts erlaubt (INTENAR = $dff01c) oder welche gesperrt sind (INTENA = $dff09a). Es ist auch möglich, anliegende Interrupts zu kontrollieren (INTREQR = $dff01e), sowie softwaremäßig "eigene" Interrupts zu provozieren (INTREQ = $dff09c). Auch Exec benutzt diese Möglichkeit bei den Funktionen Disable und Enable.

Disable	=	**-120 (Exec-Library)**

Erklärung Durch die Funktion Disable werden alle Interrupts gesperrt und der IDNestCnt erhöht.

Um die simple Routine der Disable-Funktion zu demonstrieren, folgt nun ein Auszug aus dem 2.0 ROM.

```
Disable:          ; Disable (-120)
                  ; keine Parameter

       move.w   #$4000,$dff09A ; (INTENA) Interrupts sperren
       addq.b   #1,$126(a6)    ; IDNestCnt++
       rts                     ; Ende
```

Bild 8.11: ROM-Auszug der Disable-Routine

Um die Interrupts wieder zuzulassen bzw. den Zähler zu erhöhen, benutzt man die Funktion Enable.

Enable	=	**-126 (Exec-Library)**

Erklärung Mit Hilfe der Funktion Enable wird der IDNestCnt erniedrigt. Wenn der Zähler -1 erreicht hat, werden die Interrupts wieder zugelassen.

Der ROM-Auszug verdeutlicht die Funktion der Routine:

```
Enable: ; Enable (-126)
                ; keine Parameter

       subq.b   #1,$126(a6)     ; IDNestCnt--
       bge      L123456         ; IDNestCnt >= 0 ?
       move.w   #$C000,$dff09A  ; Interrupts zulassen
L123456 rts                     ; Ende
```

Bild 8.12: ROM-Auszug der Enable-Routine

8.8.3 Das Interrupt-Demonstrationsprogramm

Als Demonstration haben wir uns für einen Screen-Blanker entschieden. Er benutzt den VB-Interrupt, um die Tastatur- und Mauseingabe zu kontrollieren, sowie einen Zähler zu erhöhen.

Hat der Zählerwert einen bestimmten Wert (Time) überschritten, werden alle DMA-Kanäle im DMACON-Register (Customchipregister) ausgeschaltet. Dadurch ist jegliche Bildschirmausgabe unterbunden. Zusätzlich wird noch die Hintergrundfarbe auf Schwarz gesetzt (COLOR00/$dff180 = 0).

Zum besseren Verständnis soll nun die Funktion der DMACON-Bits folgen:

DMACON-Register (schreiben) $dff096
DMACONR-Register (lesen) $dff002

Name	Bit	Bedeutung
SET/CLR	15	Durch das SET/CLR-Bit wird angegeben wie auf das Register zugegriffen werden soll. Das heißt 0=löschen/1=setzen
BBUSY	14	Blitter ist beschäftigt.
BZERO	13	Blitteroperationen ergaben Null (wird zum Kontrollieren von Gfx-Kollisionen benutzt).
----	12	nicht belegt
----	11	nicht belegt
BLTPRI	10	Blitter hat Priorität (CPU muß warten!)
DMAEN	09	Hauptschalter für alle DMA-Kanäle
PLEN	08	Bit-DMA (einschalten)
COPEN	07	Copper-DMA (einschalten)
BLTEN	06	Blitter-DMA (einschalten)
SPREN	05	Sprite-DMA (einschalten)
DKSEN	04	Disk-DMA (einschalten)
AUD3EN	03	Audio3-DMA (einschalten)
AUD2EN	02	Audio2-DMA (einschalten)
AUD1EN	01	Audio1-DMA (einschalten)
AUD0EN	00	Audio0-DMA (einschalten)

Unter der Abkürzung DMA (Direct Memory Access) versteht man den direkten Speicherzugriff, der durch einen DMA-Controller gesteuert wird.

```
*
* Kapitel 8
* Demonstrationsprogramm für externe Exceptions (Interrupts)
*

ExecBase        =       4
AllocMem        =       -198
FreeMem         =       -210
AddIntServer    =       -168
OpenLib         =       -552

Start:
        move.l  ExecBase,a6
        lea     IntName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)     ; Intuition-Library öffnen
        move.l  d0,IntBase
        beq     IntError

        move.l  #IntEnd-KeyInt,d0
        move.l  #$10000,d1
        jsr     AllocMem(a6)    ; Speicher belegen
        move.l  d0,d6

        add.l   #vbis_Prg-KeyInt,d0
        move.l  d0,vbis_Code    ; Adresse des Codes eintragen
        add.l   #Count-vbis_Prg,d0
        move.l  d0,vbis_Data    ; Adresse der Daten eintragen
        move.l  #50*30,Time     ; Zeit eintragen (50*30 = 30 Sec)

        ; Der Wert 50*30 ergibt sich aus dem Zeitintervall, in dem der
        ; VBI aufgerufen wird. Dies hängt mit der Frequenz der Bild-
        ; wiederholung zusammen (50Hz).
        ; 1/50*50*30 = 30 sec

        move.l  d6,a1
        lea     KeyInt,a0
        move.l  #IntEnd-KeyInt-1,d0
KeyLoop:
        move.b  (a0)+,(a1)+     ; Programmdaten kopieren
        dbf     d0,KeyLoop

        move.l  #5,d0           ; Interruptnummer (VB-Int)
        move.l  d6,a1
        jsr     AddIntServer(a6)        ; Server installieren
        moveq   #0,d0
IntError:
        rts                     ; Programm beenden


IntName:        dc.b    "intuition.library",0
                even
```

```
            cnop    0,4
*
* Programmdaten für die Interrupts
*

KeyInt:
VBInt:                  ; Interrupt-Struktur für VB-Interrupt
vbis_Succ:  dc.l    0              ;┐
vbis_Pred:  dc.l    0              ;│
vbis_Type:  dc.b    2              ;│ Node-Struktur
vbis_Pri:   dc.b    -127           ;│
vbis_Name:  dc.l    0              ;┘
vbis_Data:  dc.l    0              ; Zeiger auf die Daten
vbis_Code:  dc.l    0              ; Zeiger auf das Programm

vbis_Prg:                          ; VB-Programm:
        movem.l  a1/d0,-(a7)       ; Register retten
        move.b   $bfec01,d1        ; Taste auslesen
        cmp.b    18(a1),d1         ; neue Taste = alte Taste?
        beq      vbis_BEnd         ; Ja, dann verzweigen.
        move.b   d1,18(a1)         ; Nein, dann neue Taste speichern
        clr.l    (a1)              ; und Zähler löschen
vbis_BEnd:
        move.l   6(a1),a6          ; Zeiger auf IntBase nach a6
        move.l   68(a6),d0         ; MouseX und MouseY auslesen und
        cmp.l    10(a1),d0         ; mit alten Werten vergleichen.
        beq      vbBranch0         ; Keine Bewegung, dann weiter!
        move.l   d0,10(a1)         ; Sonst neue Position speichern
        clr.l    (a1)              ; Zähler löschen
vbBranch0:
        tst.l    (a1)              ; Counter = 0?
        bne      vbBranch1         ; Nein, dann verzweigen
        tst.w    4(a1)             ; Screen = DUNKEL?
        beq      vbBranch1         ; Nein, dann verzweigen

        move.w   #$8200,$dff096    ; DMACON setzen (DMA an)
        move.w   #0,4(a1)          ; Flag löschen
vbBranch1:
        addq.l   #1,(a1)           ; Zähler erhöhen
        move.l   (a1),d0           ; Zähler auslesen
        cmp.l    14(a1),d0         ; mit übergebenen Wert vergleichen
        blt      vbBranch2         ; noch nicht abschalten!

        move.w   #$200,$dff096     ; DMACON setzen (DMA aus)
        move.w   #1,4(a1)          ; Flag setzen
        move.l   #0,$dff180        ; Farbe auf Null (schwarz) setzen
vbBranch2:
        movem.l  (a7)+,a1/d0       ; Register restaurieren
        rts

* Datenbereich

Count:              dc.l       0      ; 00 Zähler für Zeit
Dark:               dc.w       0      ; 04 Flag für Zustand
IntBase:            dc.l       0      ; 06 Intuition-Basis
Mouse:              dc.l       0      ; 10 Mousekoordinaten
```

```
Time:               dc.l        0     ; 14 Zeit
Key:                dc.w        0     ; 18 alte Taste

IntEnd:
```

Programm 8.10: Demonstration Interrupts

8.8.4 Software-Interrupts

Im Gegensatz zu Hardware-Interrupts, die durch die Intervention eines externen Bausteins entstehen, wird der Software-Interrupt durch die Cause-Funktion erzeugt. Sicherlich stellt sich die Frage, weshalb es einen solchen softwaremäßigen Interrupt überhaupt gibt. Dies ist leicht zu erklären. Software-Interrupts haben eine niedrigere Priorität als alle anderen Interrupts. Jedoch unterbrechen sie die Bearbeitung von Tasks. Diese Eigenschaft ist wichtig für spezielle Abläufe des Multitasking-Systems.

Wie alle Hardware-Interrupts ist auch der Software-Interrupt mit einer IntVector-Struktur in der Exec-Base verankert. Diese als eine Art Server-Interrupt eingebundene Routine benötigt fünf weitere Strukturen, die sich auch in der Exec-Base befinden. Diese sogenannten SoftIntList-Strukturen für die Prioritäten -32, -16, 0, +16 und +32 bestehen aus einer List-Struktur, mit der die auszuführenden Interrupt-Strukturen zusammengefaßt werden.

SoftIntList-Struktur:

```
00    dc.l   *sh_Head          ;┐
04    dc.l   *sh_Tail          ;│
08    dc.l   *sh_TailPred      ;│  List-Struktur
12    dc.b    sh_Type          ;│
13    dc.b    sh_Pad           ;┘

14    dc.w    sh_Pad           ; Füllwort
```

***sh_Head, *sh_Tail, *sh_TailPred, sh_Type, sh_Pad**
List-Struktur, die zur Verwaltung der Interrupt-Strukturen benutzt wird.

sh_Pad
Füllwort, um Struktur auf Langwortgrenze zu erweitern.

Will man einen Software-Interrupt auslösen, so benötigt man zunächst eine IntVector-Struktur. Es bieten sich zwei Möglichkeiten an, den Interrupt einzurichten. Zum einen kann man durch eine Listen-Funktion (oder auch per Hand) den Interrupt in die Liste einer der fünf SoftIntList-Strukturen einhängen. Die andere Möglichkeit besteht darin, daß man die Cause-Funktion aufruft. Sie hängt die IntVector-Struktur in die, durch die Priorität bestimmte, Liste ein und löst mit

Hilfe des Customchip-Registers INTREQ einen Software-Interrupt aus. Diese Möglichkeit wird z.B. auch beim Message-System benutzt, wenn im mp_Flags Eintrag das Bit pa_SoftInt gesetzt wurde.

Cause	=	**-180 (Exec-Library)**

***Interrupt**	a1	<	Zeiger auf eine Interrupt-Struktur, die ausgeführt werden soll (es dürfen nur Prioritäten zwischen -32, -16, 0, +16 und +32 verwendet werden).
Erklärung			Durch die Funktion Cause kann ein Software-Interrupt ausgelöst werden.

Durch den provozierten Software-Interrupt wird die Software-Interrupt-Verwaltungsroutine ausgeführt. Sie durchläuft die Listen mit fallender Priorität und führt alle eingeklinkten Interrupt-Routinen aus.

Natürlich werden auch hier die Registerinhalte geändert.

a1	Zeiger auf Datenbereich des Interrupts
a5	Adresse des Interrupt-Programms

Bei Software-Interrupts jedoch wird, im Gegensatz zu Server-Interrupts, nicht kontrolliert, was zurückgegeben wurde, um gegebenenfalls die Abarbeitung der Interrupts zu unterbrechen.

8.9 Die Residents

Die Resident-Struktur ist eine besondere Verwaltungsstruktur. Sie wird hauptsächlich dazu benutzt, um sogenannte resetfeste Module einzubinden.

Während des Resetvorgangs werden die Speicherbereiche $F00000-$F80000 und $FC0000-$FFFFFF nach residenten Modulen durchsucht. Dabei dienen die ersten beiden Einträge der Struktur als Erkennungsmarke, da sie nie (oder sehr selten) in dieser Konstellation vorkommen. Es handelt sich dabei um den Code des ILLEGAL-Assemblerbefehls ($4afc) und ein Langwort, welches mit der Adresse des vorangegangenen ILLEGAL-Befehls belegt ist.

Resident-Struktur:

```
dc.w   $4afc          ; OpCode von illegal
dc.l   Resident       ; Zeiger auf illegal
...
```

Wurde eine solche Resident-Struktur gefunden, wird sie in die Tabelle eingetragen, deren Anfangsadresse im ExecBase-Eintrag ResModules (300) abgelegt ist. Zu einem etwas späteren Zeitpunkt werden dann die in der Tabelle zusammengefaßten Module mit der InitCode-Funktion installiert.

Die Tabelle enthält 4-Byte-Zeiger auf den Anfang der gefundenen Resident-Strukturen. Die Tabelle ist mit einem Null-Wert abgeschlossen. Sollten neue Adressen an die Tabelle gehängt werden, so kann man einen Zeiger einfügen, der auf eine neue Tabelle zeigt. Dieser Zeiger muß durch Setzen des 32. Bits gekennzeichnet werden.

```
Tabelle:
            dc.l    Resident1     ; Zeiger auf Resident-
            dc.l    Resident2     ; Struktur
            ...
            dc.l    Tabelle2 |  $80000000
                             |    ; Zeiger auf neue Tabelle
;           OR-Verknüpfung --

Tabelle2:
            dc.l    Resident3     ; Zeiger auf Resident-
            dc.l    Resident4     ; Struktur
            dc.l    0             ; Endkennung
```

Die angesprochene Initialisierung durch die InitCode-Funktion richtet sich nach den in der Struktur abgelegten Daten. Deshalb wollen wir uns zunächst den Aufbau der Struktur ansehen:

Resident-Struktur:

```
00   dc.w    rt_MatchWord    ; Erkennungscode (ILLEGAL)
02   dc.l   *rt_MatchTag     ; Adresse der Struktur
06   dc.l   *rt_EndSkip      ; Ende der Daten der Struktur
10   dc.b    rt_Flags        ; Funktions-Bits
11   dc.b    rt_Version      ; Versionsnummer der Struktur
12   dc.b    rt_Type         ; Typ der verwalteten Daten
13   dc.b    rt_Pri          ; Priorität
14   dc.l   *rt_Name         ; Zeiger auf Namen
18   dc.l   *rt_IDString     ; Identifikationszeichenkette
22   dc.l    rt_Init         ; Init-Daten/Routine
26           rt_SIZEOF
```

rt_MatchWord
Bei dem ersten Eintrag der Resident-Struktur handelt es sich um ein Wort, welches zur Erkennung der Resident-Struktur dient. Dabei wird die Bitkombination für den ILLEGAL-Befehl (rts_MatchWord = $4AFC) benutzt. Er kommt nur selten vor und ist somit als Erkennungmarke gut geeignet.

***rt_MatchTag**
Auch der zweite Eintrag der Struktur dient zur Erkennung. Dabei wird an dieser Stelle die Startadresse der Struktur erwartet (rt_MatchTag = *rt_MatchWord).

***rt_EndSkip**
Zeiger auf das Ende der Daten, die zur Resident-Struktur gehören.

rt_Flags
Durch die Bits des Eintrags rt_Flags kann unter anderem die Funktion festgelegt werden, die der Eintrag rt_Init haben soll. So enthält er normalerweise die Adresse einer Routine, die ausgeführt werden soll.

Resident-Flag	Wert	Bedeutung
rtf_AutoInit	128	Struktur benutzt die MakeLib-Funktion
rtf_ColdStart	1	Einbindung bei Kaltstart
rtf_SingleTask	2	(benutzt ab Version 2.0)

rt_Version
Versionsnummer der Daten, die von dieser Struktur verwaltet werden.

rt_Type
Durch den Typ wird festgelegt, welche Art Daten von der Resident-Struktur verwaltet werden sollen. Hier können die Werte der Knotentypen verwendet werden (z.B. nt_Library = 9).

rt_Pri
Wie das Kürzel verrät, handelt es sich hierbei um die Priorität des residenten Moduls.

***rt_Name**
Zeiger auf die Zeichenkette, die den Namen des Moduls enthält.

***rt_IDString**
Zeiger auf die Zeichenkette, die den Identifikationstext enthält.

rt_Init
Der letzte Eintrag der Resident-Struktur kann zwei verschiedene Zwecke erfüllen. Zum einen kann hier ein Zeiger auf eine Routine eingetragen werden, die zur Initialisierung angesprungen wird. Andernfalls kann hier die Adresse der Tabelle stehen, welche die Parameter für die MakeLibrary-Funktion enthalten muß. Dies hängt von der im Eintrag rt_Flags gewählten Bitkombination ab.

Wie man an Hand der Erklärungen der Einträge rt_Flags und rt_Init erkennen kann, ist es möglich, eine Library, ein De-

vice oder ein Resource direkt mit Hilfe der MakeLibrary-Funktion zu initialisieren.

Natürlich kann auch rt_Init auf eine Initialisierungsroutine zeigen, die bei der Bearbeitung der Resident-Struktur aufgerufen wird. Somit stehen dem Programmierer viele Möglichkeiten zur Auswahl.

Wählt man die erste Möglichkeit, so muß man eine Tabelle anlegen, die die Registerinhalte für den MakeLibrary-Aufruf enthalten. Da wir an dieser Stelle kein spezielles Demonstrationsprogramm erstellen wollen, verweise ich auf die Kapitel 9 und 11, in denen man an Hand einer Library und eines Devices die Verwendung der Resident-Struktur gut erkennen kann.

Für den Gebrauch der Resident-Strukturen bietet Exec folgende Funktionen an:

InitCode	=	-72 (Exec-Library)

startClass	**d0**	<	Bei startClass handelt es sich um einen Datenwert, der angibt welche Resident-Strukturen bearbeitet werden dürfen. Dabei wird startClass mit dem Eintrag rt_Flags der Resident-Struktur durch AND-Verknüpung verglichen (wenn startClass **AND** rt_Flags != 0, dann ausführen).
Version	**d1**	<	Ist die Version der Resident-Struktur niedriger als die hier angegebene, so wird diese Struktur nicht behandelt.
Erklärung			Durch die Funktion InitCode werden die residenten Module, die in einer Tabelle zusammengefaßt sind, auf welche ResModules zeigt, bearbeitet. Es werden dabei lediglich die durch die Parameter startClass und Version bestimmten Strukturen berücksichtigt. Die Behandlung der Strukturen wird durch die Funktion InitResident erledigt. Die Funktion findet Verwendung während des Resetvorgangs.

InitResident	=	-102 (Exec-Library)

segList	**d1**	<	Zeiger auf die Segment-Liste, die beim Aufruf der Funktion MakeLibray benötigt wird. Der Wert muß hier übergeben werden, da bei selbstinitialisierenden Resident-Strukturen dieser Parameter nicht übergeben werden kann.

resident a1 < Zeiger auf die zu initialisierende Resident-Struktur.

Erklärung Durch die Funktion InitResident wird die angegebene InitResident-Struktur eingerichtet. Dabei wird entweder die Funktion MakeLibrary mit den abgelegten Parametern oder eine Initialisierungsroutine aufgerufen. Dies richtet sich nach dem 7. Bit (rtf_AutoInit) des Eintrags rt_Flags der Resident-Struktur. Für nähere Informationen siehe Kapitel 9.

FindResident	=	**-96 (Exec-Library)**

name a1 < Zeiger auf eine mit Null abgeschlossene Zeichenkette.

resident d0 > Zeiger auf die Resident-Struktur mit dem angegebenen Namen oder eine Null, wenn die Resident-Struktur nicht gefunden werden konnte.

Erklärung Die Funktion FindResident sucht die Adresse einer Resident-Struktur mit dem angegebenen Namen.

8.9.1 Gibt es ein Leben nach dem Tod?

Wenn man einen Reset mit dem Tod vergleicht, so kann man diese Frage mit Ja beantworten, denn es gibt die Möglichkeit, seine Programme so zu schreiben, daß sie auch nach einem Reset noch im System bleiben.

Grundsätzlich stehen zwei Möglichkeiten zur Auswahl. Man kann sich entweder durch eine Resident-Struktur (KickTagPtr) oder mit den Reset-Vektoren (Cool-/Cold-/Warm-Capture) "über Wasser halten".

8.9.2 Überleben mit der Resident-Struktur

Wie wir aus dem vorangegangenen Kapitel wissen, können Resident-Strukturen nicht nur zum automatischen Initialisieren von Libraries, Devices oder Resources verwendet werden. Sie können auch benutzt werden, um eine Routine zu starten. Diese Möglichkeit kann man für ein resetfestes Programm benutzen.

Zusätzlich benötigt man die drei ExecBase-Einträge KickMemPtr, KickTagPtr und KickCheckSum. Sie werden bei einem Reset dazu benutzt, vom Programmierer angegebenen Speicher-

bereich sofort wieder zu belegen und Resident-Strukturen in die ResModules-Tabelle aufzunehmen.

KickMemPtr (546)

Der Eintrag KickMemPtr kann einen Zeiger enthalten, der auf eine MemEntry-Struktur zeigt, mit der festgelegt wird, welche Speicherbereiche nach einem Reset sofort wieder belegt werden sollen.

KickTagPtr (550)

Auch bei diesem Eintrag handelt es sich um einen Zeiger. Er enthält die Adresse einer Tabelle mit Resident-Strukturen, die während eines Resets ausgeführt werden sollen. Die Tabelle ist genauso aufgebaut, wie die für die Resident-Strukturen aus dem ROM (ResModules).

KickChkSum (554)

Der Eintrag KickChkSum enthält die Prüfsumme über die Werte der Resident-Tabelle (KickTagPtr) und über die Einträge der MemEntry-Strukturen (KickMemPtr). Um diese Prüfsumme zu errechnen, steht uns die Funktion SumKickData zur Verfügung.

SumKickData	=	**-612 (Exec-Library)**

Sum	**d0**	>	Neue Checksumme
Erklärung			Die Funktion SumKickData errechnet die Checksumme der Werte für die Einträge KickMemPtr und KickTagPtr.

```
*
* Kapitel 8
* Demonstrationsprogramm für residente Module
*

ExecBase      =       4
AllocMem      =       -198
SumKickData   =       -612
OpenLib       =       -552
CloseLib      =       -414
GetMsg        =       -372
ReplyMsg      =       -378
WaitPort      =       -384
OpenWindow    =       -204
CloseWindow   =       -72
DrawImage     =       -114
OpenWorkBench =       -210
```

```
Start:  move.l   ExecBase,a6      ; Zeiger auf nächstes Segment
        clr.l    Start-4          ; löschen, damit der Speicher
                                  ; nach Beendigung des Programms
                                  ; belegt bleibt.
        move.l   546(a6),RStart ; Zeiger auf MemEntry-Struktur
        move.l   #RStart,546(a6)        ; speichern und neuen Zeiger set-
zen

        move.l   550(a6),RT_second      ; Zeiger auf alte Resident-Ta-
belle
        tst.l    RT_second        ; speicher. War sie Null?
        beq      Ende             ; Ja, dann Ende.
        or.l     #$80000000,RT_second   ; Sonst Eintrag als Zeiger auf
                                  ; Resident-Tabelle kennzeichnen.
Ende:
        move.l   #RTab,550(a6)    ; Neue Tabelle eintragen

        jsr      SumKickData(a6)
        move.l   d0,554(a6)       ; Checksumme eintragen
        rts

* Resident (Programm)

        section  "",Code_C        ; neue Sektion

RStart:         dc.l    0               ; ml_Succ
                dc.l    0               ; ml_Pred
                dc.b    0               ; ml_Type
                dc.b    0               ; ml_Pri
                dc.l    0               ; ml_Name
                dc.w    1               ; ml_NumEntries
                dc.l    RStart          ; meu_Reqs/Addr
                dc.l    REnd-RStart     ; me_Length

RTab:
RT_first:       dc.l    RStruktur
RT_second:      dc.l    0

RStruktur:      illegal                 ; rt_MatchWord
                dc.l    RStruktur       ; rt_MatchTag
                dc.l    REnd            ; rt_EndSkip
                dc.b    1               ; rt_Flags
                dc.b    0               ; rt_Version
                dc.b    0               ; rt_Type
                dc.b    5               ; rt_Pri
                dc.l    RName           ; rt_Name
                dc.l    RID             ; rt_IDString
                dc.l    RInit           ; rt_Init

RID:            dc.b    "My Resident! v0.1",0
                even
RName:          dc.b    "My-Resident",0
                even

RInit:
```

```
        ; Hier schließt sich das Programm an, welches nach
        ; einem Reset ausgeführt werden soll.

REnd:
```

Programm 8.11: Demonstration residente Module

Die Priorität der eigenen Resident-Struktur sollte man nicht ohne Bedenken setzen, da von ihr die Position in der ResModules-Tabelle abhängt. Durch die Position ist auch die zeitliche Abfolge der Abarbeitung der einzelnen Resident-Strukturen festgelegt. So kann man z.B. noch nicht auf die Intuition-Library zugreifen, wenn man eine Priorität > 10 gewählt hat, da sie noch nicht initialisiert worden ist. Außerdem sollte man die Priorität immer höher als -60 ansetzen, da sonst die Resident-Struktur nicht behandelt würde, weil der Resident "strap" (mit Priorität -60), der den Bootvorgang auslöst, nicht mehr zurückkehrt.

Welche residenten Module des Systems welche Prioritäten besitzen, soll die folgende Tabelle zeigen. Dabei sind wir wiederum von der Version 2.0 ausgegangen. Man kann aber mit einem der bekannten System-Monitoren Xoper, ARTM oder mit dem auf der Programmdiskette vorliegenden Programm (PrintResident, Prg_8_13.s) die residenten Module auflisten, um sich einen Überblick über die Prioritäten zu verschaffen.

Name	Pri	Typ
expansion.library	110	Library
exec.library	105	Library
diag.init	105	Unknown
utility.library	103	Library
potgo.resource	100	Resource
cia.resource	080	Resource
FileSystem.resource	080	Resource
disk.resource	070	Resource
misc.resource	070	Resource
graphics.library	065	Library
gameport.device	060	Device
timer.device	050	Device
battclock.resource	045	Resource
keyboard.device	045	Device
battmem.resource	044	Resource
keymap.library	040	Library
input.device	040	Device
layers.library	031	Library
ramdrive.device	025	Device
trackdisk.device	020	Device
intuition.library	010	Library
alert.hook	005	Unknown
console.device	005	Device
mathieeesingbas.library	000	Library
syscheck	-035	Unknown
romboot	-040	Unknown

bootmenu	-050	Unknown
strap	-060	Unknown
filesystem	-081	Unknown
ramlib	-100	Unknown
audio.device	-120	Device
dos.library	-120	Library
workbench.task	-120	Task
gadtools.library	-120	Library
icon.library	-120	Library
mathffp.library	-120	Library
workbench.library	-120	Library
con-handler	-121	Unknown
shell	-122	Unknown
ram-handler	-123	Unknown

8.9.3 Überleben mit den Reset-Vektoren

Insgesamt gibt es drei sogenannte Reset-Vektoren. Diese Vektoren sind in der Exec-Base-Struktur untergebracht und können die Adressen von Routinen enthalten, die während des Resetablaufs aufgerufen werden sollen. Ist jedoch die Exec-Base-Struktur ungültig (Checksumme nicht korrekt), werden die Vektoren nicht berücksichtigt.

ColdCapture (42)
Der ColdCapture ist der erste der Vektoren, dessen Routine ausgeführt wird. Dabei muß man jedoch beachten, daß man nichts auf dem Stack ablegen darf, da er zu diesem Zeitpunkt noch nicht initialisiert ist. Deshalb muß man auch die Reset-Routine über "jmp (a5)" verlassen und kann nicht die RTS-Anweisung benutzen. Sie würde die Rücksprungadresse vom (noch nicht installierten) Stack holen. Zusätzlich zu der in a5 übergebenen Rücksprungadresse wird in a0 die Adresse der ColdCapture-Routine übergeben. Zu beachten ist noch, daß, bevor die Routine angesprungen wird, der Eintrag ColdCapture in der ExecBase gelöscht wird.

CoolCapture (46)
Der CoolCapture-Vektor wird nach der ColdCapture-Routine angesprungen. Dabei ist der Stack schon initialisiert. Die Routine kann also mit einem normalen "rts" verlassen werden, da sie durch ein "jsr (a0)" aufgerufen wurde.

WarmCapture (50)
Der letzte der Reset-Vektoren, der nach der Behandlung der residenten Module angesprungen wird, kann eigentlich nie erreicht werden. Dies liegt an der Resident-Struktur BootStrap (Priorität -60), die nicht wiederkehrt. Abgesehen davon wird der WarmCapture-Vektor bei der Reset-Routine des KickStart ROMs 2.0 ohnehin nicht mehr berücksichtigt.

Wenn man den Zeiger einer dieser Vektoren "verbogen" hat, muß man allerdings die Checksumme über diese Vektoren neu berechnen und im Eintrag LowMemChkSum ablegen. Die folgende

kleine Routine berechnet die Checksumme neu und trägt sie in den Eintrag LowMemChkSum ein.

```
        ...
        move.l   4,a6           ; Basisadresse der ExecLib
        lea      34(a6),a0      ; Adresse von SoftVer
        moveq    #0,d1          ; d1 löschen
        move.w   #22,d0         ; 23 Durchläufe

Loop:   add.w    (a0)+,d1       ; Werte addieren
        dbf      d0,Loop        ; d0 mal wiederholen

        not.w    d1             ; Wert negieren
        move.w   d1,82(a6)      ; neuen Wert in ChkSum
                                ; eintragen
        ...
```

Bild 8.13: Berechnung der LowMemChkSum von Exec

Natürlich soll auch hier das Demonstrationsprogramm nicht fehlen. Zunächst wird Speicher für das Vektor-Programm belegt, in den dann das Programm kopiert wird. Danach wird die Adresse in die ExecBase-Struktur eingetragen und die Checksumme neu berechnet. Während eines Resets wird dann das Vektor-Programm ausgeführt. Es belegt erneut den Speicherbereich, in dem es steht, damit es nicht vom System als frei eingestuft und überschrieben wird.

```
*
* Kapitel 8
* Demonstrationsprogramm für den CoolCapture-Vektor
*

ExecBase      =       4
AllocMem      =       -198
AllocAbs      =       -204

Start:  move.l   ExecBase,a6
        move.l   #VPEnd-VPStart,d0
        moveq    #0,d1
        jsr      AllocMem(a6)   ; Speicher für Programm besorgen
        move.l   d0,d6

        lea      VPStart,a0
        move.l   d6,a1
        move.l   #VPEnd-VPStart-1,d0
CopyLoop:
        move.b   (a0)+,(a1)+    ; Programmdaten kopieren
        dbf      d0,CopyLoop

        move.l   d6,46(a6)      ; Vektor eintragen
```

```
        lea       34(a6),a0
        moveq     #0,d1
        move.w    #22,d0
Loop:   add.w     (a0)+,d1        ; Checksumme neu berechnen
        dbf       d0,Loop
        not.w     d1
        move.w    d1,82(a6)       ; und eintragen
        rts

* Vektor-Programm

VPStart:
        move.l    ExecBase,a6
        lea       VPStart,a1
        move.l    #VPEnd-VPStart,d0
        jsr       AllocAbs(a6)    ; Speicher neu belegen
        tst.l     d0
        beq       Error

        ; ... Programm ...

        move.l    #$fffff,d0
VPLoop: move.w    d0,$dff180      ; Hintergrundfarbe setzen
        sub.l     #1,d0
        bne       VPLoop

        rts

Error:  move.l    #$4000,d0
VPELoop:
        move.w    #$d00,$dff180   ; Hitergrundfarbe rot
        dbf       d0,VPELoop

        move.l    #0,46(a6)       ; Vektor löschen
        rts
VPEnd:
```

Programm 8.12: Demonstration CoolCapture-Vektor

8.9.4 Die Kickstart-Resetroutine

Bevor wir zur ExecBase-Struktur kommen, wollen wir uns den Resetvorgang ansehen. Dabei sind wir von der KickStart- Version 2.0 ausgegangen. Wenn auch einige Unterschiede zu den vorangegangenen Versionen bestehen, so sind die Grundzüge gleich.

```
*
*             Amiga Exec KickStart 2.0 Reset-Routine
*

LF80000       dc.w    1114            ; Erkennung

              JMP     LF800D2         ; Reset-Routine anspringen

              dc.w    0
              dc.w    -1
              dc.w    $25
              dc.w    $af
              dc.w    $25             ; Version
              dc.w    $84             ; Reversion
              dc.l    -1

LF80018       dc.b    "exec 37.132 (23.5.91)",$0d,$0a,0

              dc.b    $0a,"AMIGA ROM Operating System "
              dc.b    $0a,"Copyright 1985,1986,1987,1988,1989,1990,"
              dc.b    "1991 Commodore-Amiga, Inc."
              dc.b    $0a,"All Rights reserved.",$0a,$00

LF800A6       dc.b    "exec.library",0,0,0,0

; Exec-Resident-Modul:

LF800B6       dc.w    $4afc           ;  rt_MatchWord
              dc.l    LF800B6         ; *rt_MatchTag
              dc.l    LF83BFC         ; *rt_EndSkip
              dc.b    $02             ;  rt_Flags
              dc.b    $25             ;  rt_Version
              dc.b    $09             ;  rt_Type
              dc.b    $69             ;  rt_Pri
              dc.l    LF800A6         ; *rt_Name
              dc.l    LF80018         ; *rt_IdString
              dc.l    LF80420         ; *rt_Init

              reset

; Ab hier beginnt die Reset-Routine

LF800D2 LEA      $400,A7        ; Stackzeiger setzen

        LEA      $F80000,A0     ; Anfangsadresse des Moduls
        MOVEQ    #-1,D1         ; Zähler 1 = -1 ($FF)
        MOVEQ    #1,D2          ; Zähler 2 =  1
        MOVEQ    #0,D5          ; Sum-Register löschen
LF800E2 ADD.L    (A0)+,D5       ;
LF800E4 BCC.S    LF800E8        ; Carry-Flag clear ?
        ADDQ.L   #1,D5          ; sonst d5++
LF800E8 DBF      D1,LF800E2     ; Schleife 1
        DBF      D2,LF800E2     ; Schleife 2
```

```
        LEA     LF80000(PC),A0  ; Anfangsadresse des Moduls
LF800F4 LEA     $F00000,A1      ; a1 = $F00000
        CMPA.L  A0,A1           ; Modul fängt bei $F00000 an ?
        BEQ.S   LF8010C         ; ja, dann nicht initialisieren

; Externes Modul einbinden

        LEA     LF8010C(PC),A5  ; Rücksprungadresse auf Stack legen
        CMPI.W  #$1111,(A1)     ; Modul-Kennung gefunden ?
        BNE.S   LF8010C         ; Nein, dann weiter
        JMP     2(A1)           ;  Modul anspringen

; Hardware-Register initialisieren

LF8010C CLR.B   $BFE001         ; LED einschalten (hell)
LF80112 MOVE.B  #3,$BFE201      ; 8520 in Ausgabemodus versetzen

        LEA     $DFF000,A4      ; Basisadresse der Customchips
        MOVE.W  #$7FFF,D0       ; Wert zum Interrupt löschen
        MOVE.W  D0,$9A(A4)      ; INTENA löschen
        MOVE.W  D0,$9C(A4)      ; INTREQ löschen
        MOVE.W  D0,$96(A4)      ; DMACON löschen
        MOVE.W  #$174,$32(A4)   ; serrielles Kontrollregister
        MOVE.W  #$200,$100(A4)  ; BPLCON0
        MOVE.W  #0,$110(A4)     ; BPL1DAT
        MOVE.W  #$444,$180(A4)  ; Hintergrundfarbe auf grau stellen

        MOVE.W  #$F00,D0        ; Parameter für F.beh.rout. ablegen
        NOT.L   D5              ; d5 invertieren und verzweigen,
        BNE     LF803B6         ; wenn d5 nicht Null

; Vektoren 2-47 auf Alert-Routine richten

        MOVEA.W #8,A0           ; Adresse der Vektortabelle
        MOVE.W  #$2D,D1         ; Zählerregister einrichten (45)
        LEA     LF8039E(PC),A1  ; Zeiger auf Fehlerbehand.routine
LF8015E MOVE.L  A1,(A0)+        ; Zeiger in Vektortabelle eintragen
        DBF     D1,LF8015E      ; d1 > wiederholen

; Vektoren kontrollieren

        MOVE.W  #$F0,D0         ; Parameter für eventuellen Aufruf
        MOVE.W  #$2D,D1         ; der Fehlerroutine einstellen
LF8016C CMPA.L  -(A0),A1        ; zeigt der Vektor auf die Fehler-
        BNE     LF803B6         ; behandlungroutine ? Nein, Fehler
        DBF     D1,LF8016C      ; d1 > wiederholen

; Register d2-7 löschen

        MOVEQ   #0,D2           ;┐
        MOVEQ   #0,D3           ;│
        MOVEQ   #0,D4           ;│ Register
        MOVEQ   #0,D5           ;│ löschen
        MOVEQ   #0,D6           ;│
        MOVEQ   #0,D7           ;┘
```

```
; Kontrollieren ob es sich um einen Kaltstart handelt

        MOVE.L   4.W,D1          ; Basisadresse auslesen
        MOVEA.L  D1,A6           ; nach a6 kopieren
        BTST     #0,D1           ; 0. Bit testen
        BNE.S    LF801C4         ; ungleich dann verzweigen

; ChkBase-Eintrag kontrollieren, eventuell neu einrichten

        ADD.L    $26(A6),D1      ; ChkBase-Eintrag auslesen
        NOT.L    D1              ; invertieren
        BNE.S    LF801C6         ; Ergebnis Null ?

; Prüfsumme berechnen

        LEA      $22(A6),A0      ; Anfang der Varible
        MOVEQ    #$18,D0         ; Anzahl der Worte, die addiert
LF8019C ADD.W    (A0)+,D1        ; werden sollen.
        DBF      D0,LF8019C      ; Schleife
        NOT.W    D1              ; Wert invertieren wenn nicht Null,
        BNE.S    LF801C6         ; dann verzweigen (Kaltstart)

; ColdCapture-Vektor Behandlung

        MOVE.L   $2A(A6),D0      ; ColdCapture-Vektor auslesen
        BEQ.S    LF801B8         ; nicht gesetzt? Dann überspringen.
        MOVEA.L  D0,A0           ; Sonst Adresse nach a0 und Rück-
        LEA      LF801B8(PC),A5  ; sprungadresse nach a5 legen.
        CLR.L    $2A(A6)         ; Alten Vektor löschen und ver-
        JMP      (A0)            ; zweigen

LF801B8 MOVEM.L  $222(A6),D2-4   ; KickMemPtr,KickTagPtr,KickChkSum,
        MOVEM.L  $2A(A6),D5-7    ; ColdCapture,CoolCapture und
                                 ; WarmCapture auslesen

; Prozessor-Typ ermitteln und den Wert für AttnFlags erstellen

LF801C4 SUBA.L   A6,A6           ; a6 löschen
LF801C6 MOVEA.L  A6,A5           ; a5 löschen
        BSR      $F80B30         ; Prozessor ermitteln (AttnFlags)
        MOVEA.L  D0,A2           ; AttnFlags nach a2

; Speicher

        SUBA.L   A0,A0           ; a0 löschen
        MOVEA.L  (A0),A1         ; Wert von Adresse 0 nach a1
        CLR.L    (A0)            ; Adresse Null löschen
        SUBA.L   A3,A3           ; a3 löschen
        MOVE.L   #$F2D4B689,D1   ; d1 mit $F2D4B689 laden
        BRA.S    LF801E0         ; in Schleife einspringen
LF801DE MOVE.L   D0,(A3)         ; Adresse, auf die a3 zeigt mit
LF801E0 LEA      $4000(A3),A3    ; Wert von d0 laden, a3 += $4000
        CMPA.L   #$200000,A3     ; $200000 = A3
        BEQ.S    LF801FA         ; dann verzweigen
        MOVE.L   (A3),D0         ; Wert auslesen
        MOVE.L   D1,(A3)         ; neuen Wert eintragen
```

```
        NOP                     ; no operation
        CMP.L    (A0),D1        ; (A0) = D1
        BEQ.S    LF801FA        ; ja, dann verzweigen
        CMP.L    (A3),D1        ; (A3) = D1
        BEQ.S    LF801DE        ; ja, dann verzweigen

LF801FA MOVE.L   D0,(A3)        ; Wert von D0 eintragen
        MOVE.L   A1,(A0)        ; Wert von Adresse 0 eintragen
        MOVE.L   D2,-(A7)       ; D2 retten

        LEA      $400.W,A0      ; Zeiger auf MemList
        LEA      LF8031C(PC),A1 ; Name ("chip memory")
        MOVE.L   A3,D0          ; obere Grenze
        MOVE.L   A0,D1          ; untere Grenze
        SUB.L    D1,D0          ; Länge des Speicherbereichs
        MOVE.W   #$303,D1       ; Attribute
        MOVEQ    #-$A,D2        ; Priorität des Speicherbereichs
        BSR      $F81F32        ; > AddMemList

        MOVE.L   (A7)+,D2       ; D2 restaurieren
        LEA      $400.W,A0      ; FreeList
        MOVE.L   #$57C,D0       ; ByteSize
        BSR      $F81C02        ; > Allocate
        MOVEA.L  D0,A6          ; Pointer nach a6
        SUBA.W   #$FCE8,A6      ; Basisadresse errechnen
        MOVE.L   A6,4.W         ; Basisadresse eintragen
```

; Library-Struktur löschen ($98 Longs)

```
        MOVEA.L  A6,A0          ; Basisadresse nach a0
        MOVE.W   #$98,D1        ; Anzahl der Longs, die gelöscht
LF80238 CLR.L    (A0)+          ; werden sollen
        DBF      D1,LF80238     ; Schleife
```

; Datenbereich der Exec-Base einrichten

```
        MOVE.W   LF8000E(PC),$22(A6)   ; SoftVer eintragen
        MOVE.W   A2,$128(A6)    ; AttnFlags eintragen
        MOVE.L   A3,$3E(A6)     ; MaxLocMem
        MOVE.L   A5,$26(A6)     ; ChkBase
        MOVEM.L  D2-4,$222(A6)  ; KickMem-, KickTagPtr, KickChkSum
        MOVEM.L  D5-7,$2A(A6)   ; Cold-/Cool-/WarmCapture
```

; Überprüfen ob es einen Alert gab. Wenn ja, dann Daten in ExecBase
; eintragen, sonst -1 eintragen.

```
        MOVEQ    #-1,D6         ; d6 mit -1 initialisieren
        CMPI.L   #$48454C50,0.W ; steht "HELP" ab Adresse 0?
        BNE.S    LF80272        ; nein, dann -1 eintragen >
        MOVEM.L  $100.W,D6-7    ; ja, dann Alert-Daten nach d6-7
        BSET     #$1F,D6        ; $1F. Bit setzen

LF80272 MOVEM.L  D6-7,$202(A6)  ; LastAlert-Werte eintragen
```

; Funktionstabelle aufbauen

```
        MOVEA.L  A6,A0            ; Adresse für Sprungtabelle
        LEA      $F81F84(PC),A1   ; Zeiger auf Vektortabelle
        MOVEA.L  A1,A2            ; Offsetwerte relativ zur Vektortab
        BSR      $F81AD0          ; > MakeFunction
        MOVE.W   D0,$10(A6)       ; lib_NegSize eintragen
        MOVE.W   #$264,$12(A6)    ; lib_PosSize eintragen
```

; Listenköpfe für LibList und MemList initialisieren

```
        LEA      $17A(A6),A0      ; Adresse des Listenkopfes
        MOVE.L   A0,8(A0)         ; lh_TailPred = *lh_Head
        ADDQ.L   #4,A0            ; a0 += 4
        CLR.L    (A0)             ; lh_Tail = 0
        MOVE.L   A0,-(A0)         ; lh_Head = *lh_Tail

        LEA      $142(A6),A0      ; Adresse des Listenkopfes
        MOVE.L   A0,8(A0)         ; lh_TailPred = *lh_Head
        ADDQ.L   #4,A0            ; a0 += 4
        CLR.L    (A0)             ; lh_Tail = 0
        MOVE.L   A0,-(A0)         ; lh_Head = *lh_Tail
```

; Node des belegten Speichers in MemList einfügen

```
        LEA      $400.W,A1        ; Adresse der Node-Struktur
        BSR      $F81904          ; > Enqueue
```

; MaxExtMem Wert berechnen

```
        LEA      $C00000,A0       ; Parameter 1
        LEA      $DC0000,A1       ; Parameter 2
        BSR      LF80328          ; testen
        MOVE.L   A4,$4E(A6)       ; MaxExtMem
```

; Ext-Memory in MemListe aufnehmen

```
        LEA      $C00000,A0       ; a0 = Basisadresse des Speichers
        LEA      LF80321(PC),A1   ; a1 = Name des Speicherbereichs
        MOVE.L   A4,D0            ; MaxExtMem nach d0
        BEQ.S    LF802E2          ; nicht vorhanden, dann verzweigen
        MOVE.L   A0,D1            ; Basisadresse nach d1
        SUB.L    D1,D0            ; d0-d1 = d0 = Länge des Bereichs
        MOVE.W   #$305,D1         ; d1 = Attibute des Speichers
        MOVEQ    #-5,D2           ; d2 = Priorität
        BSR      $F81F26          ; > AddMemList
```

; Tabelle der Residenten Module einrichten

```
LF802E2 MOVE.W   #$888,$DFF180    ; Hintergrundfarbe auf grau stellen
        LEA      LF80308(PC),A0   ; Zeiger auf Suchbereichtabelle
        BSR      $F80D22          ; Resident-Liste erzeugen
        MOVE.L   D0,$12C(A6)      ; Zeiger in ResModules eintragen
```

```
; Residentes Modul mit der StartClass 2 ausführen. (Exec-Resident)

        MOVEQ   #2,D0           ; startClass
        MOVEQ   #0,D1           ; Version
        JSR     -$48(A6)        ; > InitCode

; Sollte ein Fehler bei der Behandlung der Exec-Resident-Struktur
; auftreten, wird die Hintergrundfarbe auf lila gestellt und die
; Abarbeitung angehalten.

        MOVE.W  #$F0F,$DFF180   ; Hintergrund lila einfärben
LF80306 BRA.S   LF80306         ; tote Schleife

; Tabelle der Adressbereiche, in denen nach Resident-Strukturen gesucht
; werden soll

LF80308 dc.l    $F80000         ; Speicherbereich 1
        dc.l    $1000000        ;

        dc.l    $F00000         ; Speicherbereich 2
        dc.l    $F80000         ;

        dc.l    -1              ; Endkennung

LF8031C dc.b    "chip memory",0


; MaxExtMem berechnen

LF80328 MOVEA.L A0,A4           ;
        ADDA.L  #$40000,A0      ;
LF80330 MOVEA.L A4,A2           ;
        ADDA.L  #$40000,A2      ;
        MOVE.W  #$3FFF,-$F66(A2);
        TST.W   -$FE4(A2)       ;
        BNE.S   LF80352         ;
        MOVE.W  #$BFFF,-$F66(A2);
        CMPI.W  #$3FFF,-$FE4(A2);
        BEQ.S   LF8038A         ;
LF80352 MOVE.W  #0,-$F66(A0)    ;
        MOVE.L  #$F2D4,D1       ;
        MOVE.W  D1,-$F66(A2)    ;  Berechnungsroutine des
        CMP.W   -$F66(A2),D1    ;  MaxExtMem-Wertes, auf die
        BNE.S   LF80390         ;  wir nicht näher eingehen wollen.
        CMP.W   -$F66(A0),D1    ;
        BEQ.S   LF80380         ;
LF8036E MOVE.L  #$B698,D1       ;
        MOVE.W  D1,-$F66(A2)    ;
        CMP.W   -$F66(A2),D1    ;
        BNE.S   LF80390         ;
        BRA.S   LF80384         ;
LF80380 CMPA.L  A0,A2           ;
        BEQ.S   LF8036E         ;
LF80384 MOVEA.L A2,A4           ;
LF80386 CMPA.L  A4,A1           ;
        BHI.S   LF80330         ;
```

```
LF8038A MOVE.W   #$7FFF,-$F66(A2);|
LF80390 SUBA.L   #$40000,A0      ;|
        CMPA.L   A0,A4           ;|
LF80398 BNE.S    LF8039C         ;|
        SUBA.L   A4,A4           ;┘
LF8039C RTS                      ; Routine beenden
```

; Fehlerbehandlungsroutine für die Vektoren 2-47

```
LF8039E MOVEM.L  D0-7/A0-7,$180.W      ; Register retten
        LEA      $1C0.W,A0       ; Zeiger auf Datenbereich setzen
        MOVE.L   USP,A1          ; User-Stack-Pointer nach a1
        MOVE.L   A1,(A0)+        ; USP im Speicher ablegen
        MOVE.L   (A7),(A0)+      ; Rücksprungadressen vom Stackwerte
        MOVE.L   4(A7),(A0)+     ; im Datenbereich ablegen
        MOVE.W   #$FE5,D0        ; Code für Hintergrundfarbe
```

; "normale"-Fehlerbehandlungsroutine

```
LF803B6 LEA      $DFF000,A4      ; Basisadresse der Customchips
        MOVE.W   #$200,$100(A4)  ; Bitplane Kontrollregister setzen
        MOVE.W   #0,$110(A4)     ; Bitplane Daten 1 löschen
        MOVE.W   D0,$180(A4)     ; übergebene Hintergr.farbe setzen
```

; 10 mal LED blinken lassen

```
        MOVEQ    #$A,D1          ; Schleifenzähler d1 einrichten

        MOVEQ    #-1,D0          ; Schleifenzähler d0 einrichten
LF803D0 BSET     #1,$BFE001      ; LED ausschalten (dunkel)
        DBF      D0,LF803D0      ; Schleife (ca. 0.15 sec)

        LSR.W    #2,D0           ; d0 vorbereiten
LF803DE BCLR     #1,$BFE001      ; LED anschalten (hell)
        DBF      D0,LF803DE      ; Schleife

        DBF      D1,LF803D0      ; Schleife 10 mal durchlaufen

        MOVE.L   #$15000,D0      ; Schleifenzähler initialisieren
LF803F4 MOVE.W   #0,$DFF180      ; Hintergrund schwarz einfärben
        SUBQ.L   #1,D0           ; d0--
        BGT.S    LF803F4         ; Schleife (ca. 0.15 sec)

        MOVE.W   #$4000,$DFF09A  ; Interrupt-Enabel-Register setzen
        BRA      LF80CC8         ;
```

; Tabelle der Speicherbereiche

```
LF8040C dc.l     $F80000         ; Anfang
        dc.l     $1000000        ; Ende

        dc.l     $F00000         ; Anfang
        dc.l     $F80000         ; Ende

        dc.l     -1              ; Endmarkierung
```

```
;
; Ab hier steht die Initialisierungsroutine der Exec-Resident-Struktur.
; Sie übernimmt die weitere Initialisierung des ExecBase-Datenbereichs
; und richtet unter anderem den Exec-Task ein.
;

        LEA       $DFF000,A0      ; Basisadresse der Customchips
        MOVE.W    #$AAA,$180(A0)  ; Hintergrund hellgrau einfärben


; Jetzt wird erneut eine Liste aller residenten Module erzeugt. Dabei
; werden jedoch auch die, im Eintrag KickTagPtr vermerkten, Module
; berücksichtigt.

        LEA       LF8040C(PC),A0  ; Zeiger auf Speicherbereichtabelle
        BSR       $F80CDC         ; Tabelle der residenten Module er-
        MOVE.L    D0,$12C(A6)     ; stellen & in ResModules eintragen

; Speicher für die Exec-Base belegen.

        MOVE.L    #$57C,D0        ; Größe des Speicherblocks
        MOVE.L    #$10001,D1      ; Attribute des Speichers
        JSR       -$C6(A6)        ; > AllocMem
        TST.L     D0              ; Speicher erhalten ?
        BEQ       LF8051A         ; Nein, dann verzweigen
        MOVEA.L   D0,A5           ; Adresse nach a5
        SUBA.W    #$FCE8,A5       ; Basis errechnen

; Daten der Library-Struktur in die Exec-Base übertragen

        LEA       8(A5),A1        ; Zieladresse setzen
        LEA       LF80744(PC),A0  ; Zeiger auf Daten nach a0
        MOVEQ     #$C,D0          ; Anzahl Worte die übertragen
LF8045E MOVE.W    (A0)+,(A1)+     ; werden sollen
        DBF       D0,LF8045E      ; > wiederholen

; Funktiostabelle anlegen

        MOVEA.L   A5,A0           ; Basisadresse nach a0
        LEA       $F81F84(PC),A1  ; Zeiger auf Vektortabelle
        MOVEA.L   A1,A2           ; Basisadresse für Offsetwerte
        JSR       -$5A(A6)        ; > MakeFunctions

; Weitere Daten in die Exec-Base eintragen

        MOVE.W    D0,$10(A5)      ; lib_NegSize eintragen
        MOVE.W    $22(A6),$22(A5)       ; SoftVer übernehmen
        MOVE.L    $3E(A6),$3E(A5)       ; MaxLocMem übernehmen
        MOVE.L    $4E(A6),$4E(A5)       ; MaxExtMem übernehmen
        MOVEM.L   $202(A6),D0-1   ; LastAlert-Daten
        MOVEM.L   D0-1,$202(A5)   ;    übernehmen
        MOVE.W    $128(A6),$128(A5)     ; AttnFlags übernehmen
        MOVE.L    $12C(A6),$12C(A5)     ; ResModuls Zeiger übernehmen
        MOVEM.L   $222(A6),D0-2   ; KickMemPtr-, KickTagPtr- und
        MOVEM.L   D0-2,$222(A5)   ; KickChkSum-Werte übernehmen
        MOVE.L    $2E(A6),$2E(A5)       ; CoolCapture-Vektor übernehmen
```

```
; Nun werden die Einträge der "alten" System-Listen in die neuen
; eingetragen

        LEA      $142(A6),A2    ; Adresse der MemList-Struktur alt
        LEA      $142(A5),A3    ; Adresse der MemList-Struktur neu
        MOVEA.L  (A2),A0        ; Zeiger auf ersten Knoten nach a0
        MOVE.L   A0,(A3)        ; Adresse in neuen Kopf eintragen
        MOVE.L   A3,4(A0)       ; neuen Vorgänger setzen
        MOVEA.L  8(A2),A0       ; Zeiger auf letzten Knoten nach a0
        MOVE.L   A0,8(A3)       ; Adresse in neuen Kopf eintragen
        MOVE.L   A3,(A0)        ; Nachfolger auf den ln_Tail Ein-
        ADDQ.L   #4,(A0)        ; trag des Kopfes setzen

        LEA      $17A(A6),A2    ; Adresse der LibList-Struktur alt
        LEA      $17A(A5),A3    ; Adresse der LibList-Struktur neu
        MOVEA.L  (A2),A0        ; Zeiger auf ersten Knoten nach a0
        MOVE.L   A0,(A3)        ; Adresse in neuen Kopf eintragen
        MOVE.L   A3,4(A0)       ; neuen Vorgänger setzen
        MOVEA.L  8(A2),A0       ; Zeiger auf letzten Knoten nach a0
        MOVE.L   A0,8(A3)       ; Adresse in neuen Kopf eintragen
        MOVE.L   A3,(A0)        ; Nachfolger auf den ln_Tail Ein-
        ADDQ.L   #4,(A0)        ; trag des Kopfes setzen

; Stack belegen und Bereich im Datenteil der Exec-Base eintragen

        EXG      A5,A6          ; Inhalt von a5 und a6 austauschen
        MOVE.L   #$1800,D2      ; Größe des Stacks
        MOVE.L   D2,D0          ; nach d0 kopieren
        MOVE.L   #$10000,D1     ; Attribute des Speichers
        JSR      -$C6(A6)       ; > AllocMem
        MOVE.L   D0,$3A(A6)     ; SysStackLower Eintrag setzen
        ADD.L    D2,D0          ; SysStackUpper Eintrag ausrechnen
        MOVE.L   D0,$36(A6)     ; und im Datenbereich ablegen
        MOVEA.L  D0,A7          ; Stackregister auf Speicherbereich
                                ; setzen

; Alte (vorläufige) Funktionstabelle und Library-Struktur freigeben

        MOVEA.L  A5,A1          ; Basisadresse nach a1
        MOVEQ    #0,D0          ; d0 löschen
        MOVE.W   $10(A5),D0     ; lib_NegSize nach d0
        SUBA.L   D0,A1          ; Anfangsadresse ausrechnen a1-d0
        ADD.W    $12(A5),D0     ; Länge = d0 + lib_PosSize
        JSR      -$D2(A6)       ; > FreeMem

; ChkBase errechnen und neu setzen

LF8051A MOVE.L   A6,4.W         ; neue Basisadresse eintragen
        MOVE.L   A6,D0          ; neuen ChkBase-Eintrag
        NOT.L    D0             ; errechnen
        MOVE.L   D0,$26(A6)     ; und in Exec-Base ablegen

; System-Listen im Exec-Datenbereich einrichten

        LEA      LF80712(PC),A1 ; Zeiger auf Daten
LF8052A MOVE.W   (A1)+,D0       ; Offset auslesen
```

```
        BEQ.S    LF80544          ; Ende der Datentabelle erreicht?

        LEA      0(A6,D0.W),A0    ; Adresse der Liste im Datenbereich
                                  ; nach a0 und Listenkopf anlegen
        MOVE.L   A0,8(A0)         ; ln_TailPred = *ln_Head
        ADDQ.L   #4,A0            ; a0 += 4
        CLR.L    (A0)             ; ln_Tail = 0
        MOVE.L   A0,-(A0)         ; ln_Head = *ln_Tail

        MOVE.W   (A1)+,D0         ; Priorität aus der Datentabelle in
        MOVE.B   D0,$C(A0)        ; den Listenkopf kopieren
        BRA.S    LF8052A          ; Schleife
```

; Weitere Einträge der Exec-Base initialisieren

```
LF80544 LEA      $F83988(PC),A0 ; Adresse der Routine, die für die
        MOVE.L   A0,$130(A6)    ; Einträge TaskTrapCode und Task-
        MOVE.L   A0,$134(A6)    ; ExceptCode benutzt werden soll
        MOVE.L   #$F822C6,$138(A6) ; TaskExitCode setzen
        MOVE.L   #$FFFF,$13C(A6); TaskSigAlloc (Signalmaske) setzen
        MOVE.W   #$8000,$140(A6); TaskTrapAlloc (Trapmaske) setzen
        MOVEQ    #4,D0          ; Wert für Quantum und Elapsed
        MOVE.W   D0,$120(A6)    ; Quantum-Eintrag setzen
        MOVE.W   D0,$122(A6)    ; Elapsed-Eintrag setzen
```

; Vektortabelle für Ausnahmebehandlungen einrichten

```
        LEA      $F80960(PC),A0 ; Adresse des Datenbereichs nach a0
        MOVEA.L  A0,A1          ; Adresse nach a1
        MOVEA.W  #8,A2          ; Zeiger auf Bus-Error-Vektor
        BRA.S    LF80582        ; einspringen in Schleife

LF8057C LEA      0(A0,D0.W),A3  ; Adresse der Vektorrout. = a0 + d0
        MOVE.L   A3,(A2)+       ; Adresse in Tabelle eintragen
LF80582 MOVE.W   (A1)+,D0       ; Offset nach d0 laden
        BNE.S    LF8057C        ; Tabelle beendet ?
```

; Flag-Eintrag der Library-Struktur verändern

```
        BSET     #1,$E(A6)      ; libf_summing setzen

        MOVE.L   #$40C04E75,-$210(A6)  ; MOVE SR,D0 / RTS nach -$210
        MOVE.L   #$F80AD2,-$1C(A6)     ; Supervisor
        MOVE.L   #$F813F6,$230(A6)     ; Daten in ExecBaseNewReserved-
                                ; Datenbereich eintragen
```

; Untersuchung der AttnFlags

```
        MOVE.W   $128(A6),D0    ; AttnFlags auslesen
        BTST     #0,D0          ; Bit 0 testen
        BEQ.S    LF80622        ; Null, dann verzweigen

        LEA      $F80A22(PC),A0 ; Zeiger auf Routine für die
        MOVEA.W  #8,A1          ; Vektoren 2 und 3 nach a0.
        MOVE.L   A0,(A1)+       ; Vektor für Bus-Error
        MOVE.L   A0,(A1)+       ; Vektor für Address-Error
```

```
        MOVE.L   #$F80B0C,$20.W ; Vektor für Privilege-Violation

        MOVE.L   #$F80AF4,-$1C(A6)     ; Sprung für Supervisor ändern
        MOVE.L   #$42C04E75,-$210(A6)  ; Sprung für GettCC ändern

        BTST     #1,D0           ; Handelt es sich um einen MC68010?
        BEQ.S    LF80622         ; Dann verzweigen.

        MOVE.L   #$F80BE6,-$27A(A6);┐
        MOVE.L   #$F80BFC,-$280(A6);│ Änderungen für höhere Prozessor-
        MOVE.W   D0,D1             ;│ versionen.
        ANDI.W   #$18,D1           ;│
        BEQ.S    LF80616           ;│
        MOVE.L   #$F814CE,-$34(A6) ;│ Sprung für Switch ändern
        MOVE.L   #$F81526,$230(A6) ;┘ Daten ablegen in Reserved...
        BSET     #6,$129(A6)    ;

        BTST     #3,D0           ; Steht ein MC68881 zur Verfügung?
        BEQ.S    LF80616         ; Nein, dann verzweigen

        MOVE.L   #$F80C38,-$2F8(A6)    ; Vektor ändern
        DC.W     $F4F8           ;┐
LF80616 DC.W     $4E7A           ;│
        DC.W     $0002           ;│ MatheCoProz-Befehle
        DC.W     $0040           ;│
        DC.W     $0808           ;┘
```

; Library-Checksumme berechnen und Library in die Lib.-Liste aufnehmen

```
LF80622 MOVE.L   A6,A1           ; Zeiger auf Library-Base
        JSR      -$1AA(A6)       ; > SumLibrary
        MOVEA.L  A6,A1           ; Library-Base nach a1
        BSR      $F819E6         ; > AddLibrary
```

; Debugger initialisieren

```
        BSR      $F8167C         ; > Debugger initialisieren
```

; TD/ID NestCnt setzen und Interrupt-Vektoren im Exec-Datenbereich ; einrichten

```
        MOVE.W   #$FFFF,$126(A6); TD/ID NestCnt setzen
        LEA      $DFF000,A0      ; Basisadresse der Customchips
        MOVE.W   #$8200,$96(A0) ; DMACON setzen
        MOVE.W   #$C000,$9A(A0) ; INTENA setzen
        BSR      $F82D12         ; Interrupts einrichten
```

; ChkSum-Eintrag berechnen und eintragen

```
        MOVEQ    #0,D1           ; d1 löschen
        LEA      $22(A6),A0      ; Anfangsadresse der Exec-Daten
        MOVE.W   #$17,D0         ; 24 Datenwerte addieren
LF80658 ADD.W    (A0)+,D1        ; addieren
        DBF      D0,LF80658      ; Schleife
        NOT.W    D1              ; d1 invertieren und
        MOVE.W   D1,$52(A6)      ; Wert in ChkSum eintragen
```

```
; Speicher reservieren, in dem die Task-Struktur für den Exec-Task
; entstehen soll.

        LEA      LF8075E(PC),A0 ; Zeiger auf Entry-Daten
        JSR      -$DE(A6)       ; > AllocEntry
        MOVEA.L  D0,A1          ; Adresse nach a1
        TST.L    D0             ; auf Fehler testen
        BPL.S    LF80688        ; kein Fehler, dann weiter

; Fehler beim Speicherreservieren aufgetreten, dann Alert ausgeben

        MOVEM.L  D7/A5-6,-(A7)  ; Register retten
        MOVE.L   #$80018001,D7  ; Alertnummer übergeben
        MOVEA.L  4.W,A6         ; Basisadresse von Exec
        JSR      -$6C(A6)       ; > Alert
        MOVEM.L  (A7)+,D7/A5-6  ; Register restaurieren


; Task-Struktur einrichten

LF80688 MOVEA.L  $10(A1),A0     ; Zeiger auf Speicherbereich holen
        LEA      $1008(A0),A2   ; Adresse der Task-Struktur nach a2
        MOVE.L   A0,$3A(A2)     ; tc_SPLower = Stackadresse-1000
        LEA      $1000(A0),A0   ; Stackobergrenze nach a0
        MOVE.L   A0,$3E(A2)     ; tc_SPUpper = Stackadresse
        MOVE.L   A0,$36(A2)     ; tc_SPReg = Stackadresse
        MOVE.L   A0,USP         ; UserStackPointer (a7) setzen
        MOVE.W   #$100,8(A2)    ; tc_Type = 1 (nt_Task) tc_Pri = 0
        MOVE.L   #LF800A6,$A(A2); tc_Name = "exec.library"

; Listenkopf des belegten Speichers (tc_MemEntry) in der Task-Struktur
; initialisieren

        LEA      $4A(A2),A0     ; Adresse des tc_MemEntry-Eintrags
        MOVE.L   A0,8(A0)       ; ln_TailPred = *ln_Tail
        ADDQ.L   #4,A0          ; a0 += 4
        CLR.L    (A0)           ; ln_Tail = 0
        MOVE.L   A0,-(A0)       ; ln_Head = *ln_Tail

; Jetzt wird die Node-Struktur, deren Adresse im Adreßregister a1
; zwischengespeichert worden, ist in die MemEntry-Liste aufgenommen.

        JSR      -$F0(A6)       ; > AddHead

; Adresse des Task im ThisTask-Eintrag ablegen und ihn mit AddTask
; "starten"

        MOVEA.L  A2,A1          ; Adresse der Task-Struktur nach a1
        MOVE.L   A2,$114(A6)    ; ThisTask = "exec.library"-Task
        SUBA.L   A2,A2          ; a2 löschen (initialPC = 0)
        MOVEA.L  A2,A3          ; a3 löschen (finalPC = 0)
        JSR      -$11A(A6)      ; > AddTask
```

```
; Da sich der Exec-Task durch AddTask in einer der System-Listen
; befindet, er jedoch als laufender Task eingesetzt werden soll, müssen
; wir ihn dort entnehmen. Außerdem müssen wir ihn in den Eintrag
; ThisTask eintragen.

        MOVEA.L  $114(A6),A1      ; Adresse des laufenden Tasks
        MOVE.B   #2,$F(A1)        ; ts_Run = 2
        JSR      -$FC(A6)         ; > Remove

        ANDI.W   #0,SR            ; SR bearbeiten
        ADDQ.B   #1,$127(A6)      ; TDNestCnt erhöhen
        JSR      -$8A(A6)         ; > Permit

; CoolCapture-Vektor

        MOVE.L   $2E(A6),D0       ; CoolCapture-Vektor auslesen
        BEQ.S    LF806F4          ; nicht gesetzt?
        MOVEA.L  D0,A0            ; Adresse nach a0 kopieren
        JSR      (A0)             ; verzweigen zur CoolC-Routine

; Hintergrundfarbe erneut ändern und dann alle residenten Module
; abarbeiten, bei denen das ColdStart-Flag gesetzt wurde.

LF806F4 LEA      $DFF000,A0       ; Basisadresse der Customchips
        MOVE.W   #$CCC,$180(A0)   ; Hintergrund hellgrau einfärben

        MOVEQ    #1,D0            ; StartClass
        MOVEQ    #0,D1            ; Version
        JSR      -$48(A6)         ; > InitCode

; Sollte ein Fehler bei der Bearbeitung der Module auftreten, so wird
; der Hintergrund lila eingefärbt und die Abarbeitung angehalten.

        MOVE.W   #$F0F,$DFF180    ; Hintergrund lila einfärben
LF80710 BRA.S    LF80710          ; tote Schleife

; Daten zur Initialisierung der System-Listen. Dabei gibt der erste
; Datenwert den Offset von der Basisadresse aus an und der zweite den
; Typ der Liste.

LF80712         dc.w    $0150           ; Resource-List
                dc.w    $0008           ; nt_Resource

                dc.w    $015E           ; Device-List
                dc.w    $0003           ; nt_Device

                dc.w    $0188           ; Port-List
                dc.w    $0004           ; nt_MsgPort

                dc.w    $0196           ; TaskReady-List
                dc.w    $0001           ; nt_Task

                dc.w    $01A4           ; TaskWait-List
                dc.w    $0001           ; nt_Task

                dc.w    $016c           ; Interrupt-List
```

```
            dc.w    $0002           ; nt_Interrupt

            dc.w    $01b2           ; SoftInt-List (Pri -32)
            dc.w    $000b           ; nt_SoftInt

            dc.w    $01c2           ; SoftInt-List (Pri -16)
            dc.w    $000b           ; nt_SoftInt

            dc.w    $01d2           ; SoftInt-List (Pri 0)
            dc.w    $000b           ; nt_SoftInt

            dc.w    $01e2           ; SoftInt-List (Pri 16)
            dc.w    $000b           ; nt_SoftInt

            dc.w    $01f2           ; SoftInt-List (Pri 32)
            dc.w    $000b           ; nt_SoftInt

            dc.w    $0214           ; Semaphore-List
            dc.w    $000f           ; nt_SignSem

; Daten für die Library-Struktur der Exec-Library

LF80744     dc.b    $09             ;  ln_Type
            dc.b    $9c             ;  ln_Pri
            dc.l    $00f800a6       ; *ln_Name
            dc.b    $0600           ;  lib_Flags
            dc.b    $00             ;  lib_Pad
            dc.w    $0000           ;  lib_NegSize
            dc.w    $0264           ;  lib_PosSize
            dc.w    $0025           ;  lib_Version
            dc.w    $0084           ;  lib_Reversion
            dc.l    $00f80018       ; *lib_IDString
            dc.l    $00000000       ;  lib_Sum
            dc.w    $0001           ;  lib_OpenCnt
```

Bild 8.14: Die RESET-Routine

8.10 Sonderfunktionen

Nachdem wir alle wichtigen Bereiche besprochen und die relevanten Funktionen aufgeführt haben, folgt nun noch eine Reihe von Funktionen, die nicht kategorisiert werden konnten.

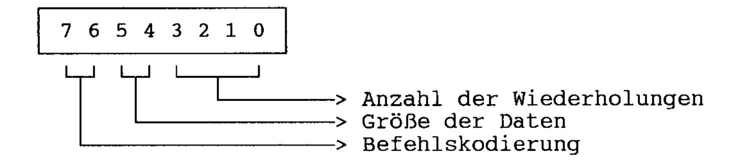
Anfangen wollen wir mit der Funktion InitStruct, die wir schon im Library-Kapitel erwähnt haben.

Die Funktion initialisiert selbständig eine Struktur. Dabei wird lediglich ein Zeiger auf einen Speicherbereich benötigt, der die Struktur aufnehmen soll, sowie eine Tabelle, nach deren kodierten Einträgen die Struktur entstehen soll.

Die Tabelle mit den kodierten Anweisungen besteht aus beliebig vielen Befehlsbytes, gefolgt von Datenwerten. Dabei ist die Länge und die Bedeutung der Daten abhängig von dem Befehlsbyte. Die Tabelle wird durch ein Null-Wort beendet.

```
...
dc.b    Befehlsbyte
dc.b    Daten,Daten,Daten,...
dc.b    Befehlsbyte
dc.w    Daten,Daten,Daten,...
dc.b    Befehlsbyte
dc.l    Daten,Daten,Daten,...
...
dc.w    0
...
```

Das Befehlsbyte läßt sich in zwei Nibbels unterteilen. Das erste Nibble (Bits 4-7 des Bytes) enthält zum einen die beiden Befehlsbits (6-7) und zwei weitere Bits (4-5), welche die Datengröße festlegen. Das untere (zweite) Nibbel (Bits 0-3 des Befehlsbytes) nimmt den Zähler für die Wiederholung auf.



Befehlskodierung (Bits 6, 7)

00 (000) Die Daten, die dem Befehlsbyte folgen, werden in die Struktur übernommen. Dabei wird die Anzahl der Werte, die übertragen werden sollen, durch den Zähler im unteren Nibbel festgelegt.

01 (064) Der nachfolgende Datenwert wird in die Struktur kopiert. Die Anzahl, wie oft dieser Wert kopiert werden soll, enthält wiederum das untere Nibbel des Befehlsbytes.

10 (128) Das, dem Befehlsbyte folgende Byte dient als Offsetwert, der zu der angegebenen Basisadresse addiert wird. Danach werden die folgenden Datenwerte an die errechnete Position kopiert. Alle weiteren Befehle beziehen sich dann auf die Adresse nach dem letzten kopierten Datenwert.

11 (192) In der Form ist dieser Befehl identisch mit dem vorangegangenen. Es werden lediglich

statt eines Offsetbytes drei Bytes als Offsetwerte benutzt.

Größe der Daten (Bits 4, 5)

00 (000) Der Befehl bezieht sich auf Langwort-Größe (das Daten-Langwort muß an einer geraden Adresse liegen).

01 (016) Der Befehl bezieht sich auf Wort-Größe (das Daten-Wort muß an einer geraden Adresse liegen).

10 (032) Der Befehl bezieht sich auf Byte-Größe.

11 (048) Diese Kodierung wird nicht benutzt.

Anzahl der Wiederholungen (Bits 0-3)

Die letzten vier Bits werden, wie schon erwähnt, als Zähler benutzt. Dabei wird der Zähler bis -1 heruntergezählt. Das bedeutet, daß die Anzahl der Wiederholungen durch **n-1** angegeben werden muß.

Zu beachten ist, daß nach dem Befehlsbyte immer ein Füllbyte folgen muß, wenn die Operation sich auf Daten bezieht, die größer sind als ein Byte (Word/Longword). Außerdem müssen alle Befehlsbytes auf einer geraden Adresse liegen.

Als Beispiel dieser relativ komplizierten Möglichkeit soll eine Struktur aufgebaut werden:

```
        dc.b    %00000011       ; Befehlsbyte
        dc.b    0               ; Füllbyte
        dc.l    0974,6222,0,1910      ; Datenwert

        dc.b    %01010100       ; Befehlsbyte
        dc.b    0               ; Füllbyte
        dc.w    4711            ; Datenwert

        dc.b    %10100001       ; Befehlsbyte
        dc.b    40              ; Offsetwert
        dc.b    46,12           ; Datenwert

        dc.b    %11000001       ; Befehlsbyte
        dc.b    0,12,03         ; 24 Bit Offsetwert
        dc.l    3344,5566       ; Datenwert
```

Bild 8.15: Beispiel für InitStruct-Strukturaufbau

Die vorangegangene Befehlstabelle legt, unter Anwendung aller zur Verfügung stehenden Kombinationsmöglichkeiten, eine Struktur mit folgendem Aufbau an.

```
0000    dc.l     0974,6222,0,1910            ;
0016    dc.w     4711,4711,4711,4711,4711    ;
0026    dcb.b    14,0                        ; 14 Null-Bytes
0040    dc.b     46,12                       ;
0042    dcb.b    4569,0                      ; 4569 Null-Bytes
4611    dc.l     3344,5566                   ;
```

Bild 8.16: Ergebnis des InitStruct-Strukturaufbaus

InitStruct	**= -78 (Exec-Library)**

initTable	**a1**	<	Zeiger auf eine Tabelle mit kodierten Befehlsworten und Daten, mit deren Hilfe die Struktur erstellt werden soll.
memory	**a2**	<	Zeiger auf den Speicherbereich, in der die Struktur erstellt werden soll.
size	**d0**	<	Die Anzahl der durch **size** angegebenen Bytes wird ab der in **memory** übergebenen Adresse gelöscht.

Erklärung

Durch die Funktion InitStruct kann mit Hilfe einer Tabelle mit kodierten Befehlsbytes und Daten, sowie der Länge und der Anfangsadresse, eine Struktur erstellt werden.

Die folgenden vier Funktionen werden hauptsächlich von dem integrierten Debugger ROM-Wack benutzt, der durch sie die Ausgabe erledigt.

RawIOInit	**= -504 (Exec-Library)**

Erklärung

Mit der Funktion RawIOInit wird die Baudrate eingestellt.

RawMayGetChar	**= -510 (Exec-Library)**

Char	**d0**	>	Gelesenes Zeichen oder -1.

Erklärung

Die Funktion testet, ob im seriellen Datenpuffer ein Zeichen anliegt. Ist dies der Fall, wird in d0 das gelesene Zeichen zurückgegeben oder -1.

RawPutChar = -516 (Exec-Library)

Char	d0	<	Zeichen, welches gesendet werden soll.
Erklärung			Das angegebene Zeichen wird über die serielle Schnittstelle gesendet.

RawDoFmt = -522 (Exec-Library)

Format	a0	<	Zeiger auf eine formatierte Zeichenkette.
Data	a1	<	Zeiger auf die Daten, die in die Zeichenkette eingefügt werden sollen.
PutProc	d0	<	Zeiger auf eine Routine, die die Daten ausgibt
PutData	d1	<	Zeiger auf eine Routine, die für das Einsetzen der Daten zuständig ist.
Erklärung			Mit Hilfe der Funktion RawDoFmt kann man eine Zeichenkette ausgeben (senden).

Alert = -108 (Exec-Library)

Parameters	a5	<	Zeiger auf den Task, von dem der Alert aufgerufen wird.
AlertNum	d7	<	Alertnummer (Fehlernummer, siehe Anhang).
Erklärung			Durch die Alert-Funktion wird ein Alert mit der angegebenen Fehlernummer erzeugt. Zusätzlich zu der Fehlernummer wird noch die Adresse des Tasks benötigt (die Fehlernummern für "Gurus" sind im Anhang aufgelistet).

SetSR = -144 (Exec-Library)

NewSR	d0	<	Neuer Wert des Statusregisters.
Mask	d1	<	Wert für die Maskierung der Werte bei der Erneuerung des Statusregisters.
OldSR	d0	>	Alter Wert des Statusregisters.
Erklärung			Durch die Funktion SetSR kann man den Wert des Statusregisters neu setzen. Dabei werden nur die durch die Maske angegebenen Bits verändert.

GetCC = -528 (Exec-Library)

Conditions	**d0**	>	Werte der Statusflags (X,N,Z,V,C).
Erklärung			Die Funktion GetCC kann benutzt werden um den Zustand der Statusflags auszulesen.

SuperState = -150 (Exec-Library)

OldSysStack	**d0**	>	Alter Systemstackzeiger.
Erklärung			Durch SuperState wird der Prozessor in den Supervisormodus versetzt. Dabei wird der alte Systemstack zurückgegeben.

UserState = -156 (Exec-Library)

OldSysStack	**d0**	<	Alter Stackzeiger
Erklärung			Durch die Funktion UserState wird der Prozessor wieder in den Usermode zurückversetzt.

Debug = -114 (Exec-Library)

Erklärung			Der Debugger ROM-Wack wird aufgerufen.

CopyMem = -624 (Exec-Library)

Source	**a0**	<	Zeiger auf Quelle.
Dest	**a1**	<	Zeiger auf Ziel.
Size	**d0**	<	Länge des Bereichs, der kopiert werden soll (Bytes).
Erklärung			Der angegebenen Datenbereich wird kopiert.

CopyMemQuick = -630 (Exec-Library)

Source	**a0**	<	Zeiger auf Quelle (muß durch 4 teilbar sein).
Dest	**a1**	<	Zeiger auf Ziel (muß durch 4 teilbar sein).

Size d0 < Länge des Bereichs, der kopiert werden soll (in Bytes, muß durch 4 teilbar sein).

Erklärung CopyMemQuick ist eine verbesserte Version des CopyMem-Befehls. Wenn der Quell- und der Zielbereich sich überschneiden, kann nur nach unten kopiert werden (a1 < a0).

Auf die Funktionen Procure, Vacate, InitSemaphore, ObtainSemaphore, ReleaseSemaphoreList, FindSemaphore, AddSemaphore sowie RemSemaphore wollen wir nicht eingehen, da ihre Anwendung auf sog. "Semaphores" zugeschnitten ist, die für System-Porgrammierer kaum interessant sind.

8.11 Die Basisstruktur der Exec-Library

Wie schon im Kapitel über Libraries erklärt wurde, kann sich, je nach Library, an die Grundstruktur ein Datenteil anschließen. Bei Exec sind hier wichtige Daten zur Verwaltung abgelegt. Auf diese Daten kann über positive Offsets zugegriffen werden.

ExecBase-Struktur:

```
000   dc.l   *ln_Succ        ;┐     ┐
004   dc.l   *ln_Pred        ;│     │
008   dc.b    ln_Type        ;│ Node│
009   dc.b    ln_Pri         ;│     │
010   dc.l   *ln_Name        ;┘     │
                                    │
014   dc.w    lib_Flags      ;      │
016   dc.w    lib_NegSize    ;      │ Library-Struktur
018   dc.w    lib_PosSize    ;      │
020   dc.w    lib_Version    ;      │
022   dc.w    lib_Revison    ;      │
024   dc.l   *lib_idString   ;      │
028   dc.l    lib_Sum        ;      │
032   dc.w    lib_OpenCnt    ;      ┘
```

; Anfang der ExecBase-Daten

```
034   dc.w    SoftVer        ; KickStart release number
036   dc.w    LowMemChkSum   ; Checksumme von 34 bis 78
038   dc.l    ChkBase        ; Sys-Ptr Komplement

042   dc.l   *ColdCapture    ; Zeiger auf ColdCapture
046   dc.l   *CoolCapture    ; Zeiger auf CoolCapture
050   dc.l   *WarmCapture    ; Zeiger auf WarmCapture

054   dc.l    SysStkUpper    ; obere Stackgrenze
```

```
058     dc.l    SysStkLower         ; untere Stackgrenze
062     dc.l    MaxLocMem           ; Obergrenze Chip-Ram

066     dc.l    DebugEntry          ; Zeiger auf ROM-Wack
070     dc.l    DebugData           ; Zeiger auf ROM-Wack Daten
074     dc.l    AlertData           ; AlertData

078     dc.l    MaxExtMem           ; Obergrenze Fast-Ram
082     dc.w    ChkSum              ; Checksumm 34 bis 78

; Interrupt-Vektoren

084     dc.l    iv_Data             ; IntVector-Struktur
088     dc.l    iv_Code             ; für seriellen Port
092     dc.l    iv_Node             ; (TBE)

096     dc.l    iv_Data             ; IntVector-Struktur
100     dc.l    iv_Code             ; für Disk-Block fertig
104     dc.l    iv_Node             ; (DskBlk)

108     dc.l    iv_Data             ; IntVector-Struktur
112     dc.l    iv_Code             ; für Soft-Interrupt
116     dc.l    iv_Node             ; (SoftInt)

120     dc.l    iv_Data             ; IntVector-Struktur
124     dc.l    iv_Code             ; für CIAA-Interrupt
128     dc.l    iv_Node             ; (Ports)

132     dc.l    iv_Data             ; IntVector-Struktur
136     dc.l    iv_Code             ; für Copper-Interrupts
140     dc.l    iv_Node             ; (Coper)

144     dc.l    iv_Data             ; IntVector-Struktur
148     dc.l    iv_Code             ; für Bildwechsel-Interrupt
152     dc.l    iv_Node             ; (VertB)

156     dc.l    iv_Data             ; IntVector-Struktur
160     dc.l    iv_Code             ; für Blitter-Interrupt
164     dc.l    iv_Node             ; (Blit)
168     dc.l    iv_Data             ; IntVector-Struktur
172     dc.l    iv_Code             ; für Audio-Kanal-Interrupt
176     dc.l    iv_Node             ; (Aud0)

180     dc.l    iv_Data             ; IntVector-Struktur
184     dc.l    iv_Code             ; für Audio-Kanal-Interrupt
188     dc.l    iv_Node             ; (Aud1)

192     dc.l    iv_Data             ; IntVector-Struktur
196     dc.l    iv_Code             ; für Audio-Kanal-Interrupt
200     dc.l    iv_Node             ; (Aud2)

204     dc.l    iv_Data             ; IntVector-Struktur
208     dc.l    iv_Code             ; für Audio-Kanal-Interrupt
212     dc.l    iv_Node             ; (Aud3)

216     dc.l    iv_Data             ; IntVector-Struktur
```

```
220     dc.l    iv_Code              ; für seriellen Port
224     dc.l    iv_Node              ; (RBF)

228     dc.l    iv_Data              ; IntVector-Struktur
232     dc.l    iv_Code              ; für Diskettensyncronisation
236     dc.l    iv_Node              ; (DskSyn)

240     dc.l    iv_Data              ; IntVector-Struktur
244     dc.l    iv_Code              ; für Externe Bausteine
248     dc.l    iv_Node              ; (Exter)

252     dc.l    iv_Data              ; IntVector-Struktur
256     dc.l    iv_Code              ; für den Master Interrupt
260     dc.l    iv_Node              ; (INTEN)

264     dc.l    iv_Data              ; IntVector-Struktur
268     dc.l    iv_Code              ; für nicht maskierbare Int.
272     dc.l    iv_Node              ; (NMI)
```

; System-Variablen

```
276     dc.l   *ThiskTask            ; Zeiger auf laufenden Task
280     dc.l    IdleCount            ; Warte-Zähler
284     dc.l    DispCount            ; Dispatch-Zähler
288     dc.w    Quantum              ; Zeitscheibengröße
290     dc.w    Elapsed              ; vergangene Zeit
292     dc.w    SysFlags             ; System-Flags
294     dc.b    IDNestCnt            ; Zähler (Interrupt Disable)
295     dc.b    TDNestCnt            ; Zähler (Task Disable)
296     dc.w    AttnFlags            ; Flags für Prozessor
298     dc.w    AttnResched          ; Flags für Reschedule
300     dc.l    ResModules           ; Adresse der Resident-List
304     dc.l    TaskTrapCode         ; Trap-Handler
308     dc.l    TaskExceptCode       ; Exception-Handler
312     dc.l    TaskExitCode         ; Exit-Routine
316     dc.l    TaskSigAlloc         ; belegte Signale des Tasks
320     dc.w    TaskTrapAlloc        ; belegte Traps
```

; Header-Strukturen der Daten

```
322     dc.l   *lh_Head              ;┐
326     dc.l   *lh_Tail              ;│ Listenheader
330     dc.l   *lh_TailPred          ;│     für
334     dc.b    lh_Type              ;│    Memory
335     dc.b    lh_Pad               ;┘

336     dc.l   *lh_Head              ;┐
340     dc.l   *lh_Tail              ;│ Listenheader
344     dc.l   *lh_TailPred          ;│     für
348     dc.b    lh_Type              ;│   Resources
349     dc.b    lh_Pad               ;┘

350     dc.l   *lh_Head              ;┐
354     dc.l   *lh_Tail              ;│ Listenheader
358     dc.l   *lh_TailPred          ;│     für
362     dc.b    lh_Type              ;│   Devices
```

```
363    dc.b    lh_Pad          ;

364    dc.l   *lh_Head         ;
368    dc.l   *lh_Tail         ;  Listenheader
372    dc.l   *lh_TailPred     ;      für
376    dc.b    lh_Type         ;   Interrupts
377    dc.b    lh_Pad          ;

378    dc.l   *lh_Head         ;
382    dc.l   *lh_Tail         ;  Listenheader
386    dc.l   *lh_TailPred     ;      für
390    dc.b    lh_Type         ;   Libraries
391    dc.b    lh_Pad          ;

392    dc.l   *lh_Head         ;
396    dc.l   *lh_Tail         ;  Listenheader
400    dc.l   *lh_TailPred     ;      für
404    dc.b    lh_Type         ;     Ports
405    dc.b    lh_Pad          ;

406    dc.l   *lh_Head         ;
410    dc.l   *lh_Tail         ;  Listenheader
414    dc.l   *lh_TailPred     ;      für
418    dc.b    lh_Type         ;  ready Tasks
419    dc.b    lh_Pad          ;

420    dc.l   *lh_Head         ;
424    dc.l   *lh_Tail         ;  Listenheader
428    dc.l   *lh_TailPred     ;      für
432    dc.b    lh_Type         ;  wait Tasks
433    dc.b    lh_Pad          ;

; SoftInterrupt

434    dc.l   *is_Head         ;
438    dc.l   *is_Tail         ;
442    dc.l   *is_TailPred     ;
446    dc.b    is_Type         ;  SoftInt (Priorität -32)
447    dc.b    is_Pad          ;
448    dc.w    is_Pad          ;

450    dc.l   *is_Head         ;
454    dc.l   *is_Tail         ;
458    dc.l   *is_TailPred     ;
462    dc.b    is_Type         ;  SoftInt (Priorität -16)
463    dc.b    is_Pad          ;
464    dc.w    is_Pad          ;

468    dc.l   *is_Head         ;
438    dc.l   *is_Tail         ;
442    dc.l   *is_TailPred     ;
446    dc.b    is_Type         ;  SoftInt (Priorität 00)
447    dc.b    is_Pad          ;
448    dc.w    is_Pad          ;
```

```
434     dc.l    *is_Head              ;┐
438     dc.l    *is_Tail              ;│
442     dc.l    *is_TailPred          ;│
446     dc.b     is_Type              ;│ SoftInt (Priorität 16)
447     dc.b     is_Pad               ;│
448     dc.w     is_Pad               ;┘

434     dc.l    *is_Head              ;┐
404     dc.l    *is_Tail              ;│
506     dc.l    *is_TailPred          ;│
510     dc.b     is_Type              ;│ SoftInt (Priorität 32)
511     dc.b     is_Pad               ;│
512     dc.w     is_Pad               ;┘
```

; Informationen

```
514     ds.l    4                     ; AlertData
530     dc.b     VBlanckFrequency     ; Freq.(Bildaufbau)
531     dc.b     PowerSupplyFrequency ; Freq.(Netzspannung)
```

; SemaphoreList

```
532     dc.l    *lh_Header            ;┐
536     dc.l    *lh_Tail              ;│ Listenkopf
540     dc.l    *lh_TailPred          ;│    für
544     dc.b     lh_Type              ;│ SemaphoreList
545     dc.b     lh_Pad               ;┘

546     dc.l     KickMemPtr           ; Zeiger auf MemList-Struktur
550     dc.l     KickTapPtr           ; Zeiger auf Resident-Tabelle
554     dc.l     KickChkSum           ; Prüfsumme (SumKickData)

558     ds.b     10                   ; ExecBaseReserved
568     ds.b     20                   ; ExecBaseNewReserved
```

***ln_Succ - lib_OpenCnt**
Wie jede Library fängt auch die Exec-Library mit einer Library-Struktur an.

SoftVer
Überarbeitungsnummer des Kickstarts.

LowMemChkSum
Hier steht der Wert der Prüfsumme, die über die Einträge SoftVer bis MaxExtMem berechnet wird. Wenn man Werte, die in diesem Bereich liegen, verändert hat, sollte man die Checksumme neu berechnen. Eine entsprechende Routine finden Sie im Abschnitt über die Reset-Vektoren.

ChkBase
Durch den ChkBase Eintrag wird überprüft, ob nach einem Reset die ExecBase-Struktur neu erstellt werden soll. Dazu wird die Addresse der ExecBase zu dem gespeicherten Wert addiert. Erhält man den Wert $FFFFFFFF (-1L), ist es nicht nötig, die Struktur neu zu installieren.

ColdCapture, CoolCapture, WarmCapture
Hier stehen drei Vektoren, die während eines Resets angesprungen werden können. Damit ist die Möglichkeit gegeben, resetfeste Programme zu realisieren.

SysStkUpper, SysStkLower
Durch SysStkUpper und SysStkLower sind die Grenzen des Supervisor-Stacks angegeben.

MaxLocMem
Maximal erreichbarer Chip-Mem-Bereich.

DebugEntry, DebugData
Durch DebugEntry ist der Anfang des integrierten Debuggers angegeben. Sinngemäß enthält DebugData den Zeiger auf den zu verwendenden Speicherbereich für die Daten des Debuggers.

AlertData
Daten des, nach einem Reset darzustellenden Alerts.

MaxExtMem
Maximal erreichbarer Bereich der Speichererweiterung.

ChkSum
ChkSum enthält eine Checksumme über die Einträge 34 bis 78. Für die Berechnung gilt die gleiche Routine wie bei LowMem-ChkSum.

IntVects 0-15
(IVTBE, IVDSKBLK, IVSOFTINT, IVPORTS, IVCOPER, IVVERTB, IV-BLIT, IVAUD0, IVAUD1, IVAUD2, IVAUD3, IVRBF, IVDISKSYNC, IVEXTER, IVINTEN, IVNMI)
IntVektor-Strukturen der folgenden Interrupts:
Serieller Port T-Buffer, Disk Block Fertig, Software Interrupt, IO-Ports und Timers, Copper, Start Vertical Blank, Blitter fertig, Audio Kanal 0 fertig, Audio Kanal 1 fertig, Audio Kanal 2 fertig, Audio Kanal 3 fertig, serieller Port R-Buffer, Disk Sync Reg, Externer Interrupt, Master Interrupt, Nicht maskierbarer Interrupt.

***ThisTask**
Zeiger auf die TaskControl-Struktur des laufenden Tasks.

IDNestCnt, TDNestCnt
Die Einträge IDNestCnt (Interrupt Disable Nesting Counter) und TDNestCnt (Task Disable Nesting Counter) dienen als Zähler für die Funktionsaufrufe Forbid und Disable. Diese Werte gelten für den laufenden Task und werden beim Taskswitching in die TaskControl-Struktur eingetragen. Dabei können die Aufrufe auch verschachtelt werden. Die Funktion des Taskswitching bzw. die Interrupts werden erst zugelassen, wenn der Zähler auf -1 steht.

AttnFlags
Durch die Kombination der Bits des Eintrags AttnFlags ist festgelegt, welcher Prozessor benutzt wird.

1	68010
2	68020
16	68881

ResModules
Zeiger auf eine Tabelle, welche die Adressen der residenten Module enthält (siehe Resident-Struktur).

MemList
Listenkopf einer Kette, in der die freien Speicherbereiche enthalten sind.

ResourceList
Listenkopf einer Kette, in der alle Resource-Strukturen enthalten sind.

DeviceList
Listenkopf einer Kette, in der alle Device-Strukturen enthalten sind.

LibList
Listenkopf einer Kette, in der alle Library-Strukturen enthalten sind.

PortList
Listenkopf einer Kette, in der alle Port-Strukturen enthalten sind.

TaskReady
Listenkopf einer Kette, in der alle Task-Strukturen enthalten sind die darauf warten, den Prozessor wieder zu übernehmen.

TaskWait
Listenkopf einer Kette, in der alle Task-Strukturen enthalten sind, die auf ein Signal eines anderen Tasks warten.

SoftInts (-32,-16,0,+16,+32)
Listenköpfe für Software-Interrupts der Prioritäten -32, -16, 0, +16 und +32.

LastAlert
Diese vier Langworte enthalten die Daten für den, nach einem Reset darzustellenden Alert.

VBlankFrequency
Der Eintrag VBlankFrequency gibt die Frequenz des Bildaufbaus an.

PowerSupplyFrequency
enthält die Frequenz der Netzspannung. Anhand dieser Frequenz wird z.B. auch zwischen 200 und 256 Punkten in der Bildschirm-Y-Auflösung (PAL/NTSC) unterschieden.

SemaphoreList
Listenkopf einer Kette, in der alle Semaphore-Strukturen enthalten sind. Semaphore-Strukturen sind Erweiterungen des Message-Systems. Da sie aber sehr selten benötigt werden, haben wir auf eine nähere Erklärung verzichtet.

KickMemPtr
Der Zeiger der im Eintrag KickMemPtr steht verweist auf eine MemList-Struktur, welche die Speicherbereiche enthält, die nach einem Reset wieder als belegt eingetragen werden sollen.

KickTagPtr
Zeiger auf eine Tabelle mit Zeigern auf Resident-Strukturen, die vom Programmierer eingesetzt werden können (siehe resetfeste Programme).

KickChkSum
Hier steht die Checksumme, die über die KickMemPtr- und KickTagPtr-Einträge von SumKickData berechnet werden kann.

ExecBaseReserved
Dieser Bereich (10 Byte) ist für Exec reserviert, um Daten zwischenzuspeichern.

ExecBaseNewReserved
Dieser Bereich (20 Byte) ist für Exec reserviert, um Daten zwischenzuspeichern.

Einige Bereiche von Exec sind recht interessant und helfen die Arbeit von Exec zu verstehen. Deshalb sollte man sich die Routinen, wenn man sich für sie interessiert, mit dem Debugger, oder besser noch in einem Buch mit dem kommentierten ROM-Listing ansehen. Die meisten Funktionen sind nämlich sehr einfach gebaut (z.B. der Forbid- bzw. Permit-Befehl).

Kapitel 9
Konstruktion einer eigenen Library

Allgemeiner Aufbau einer Library-Datei

Die MakeLibrary-Routine im Detail

Bestandteile unserer eigenen Library

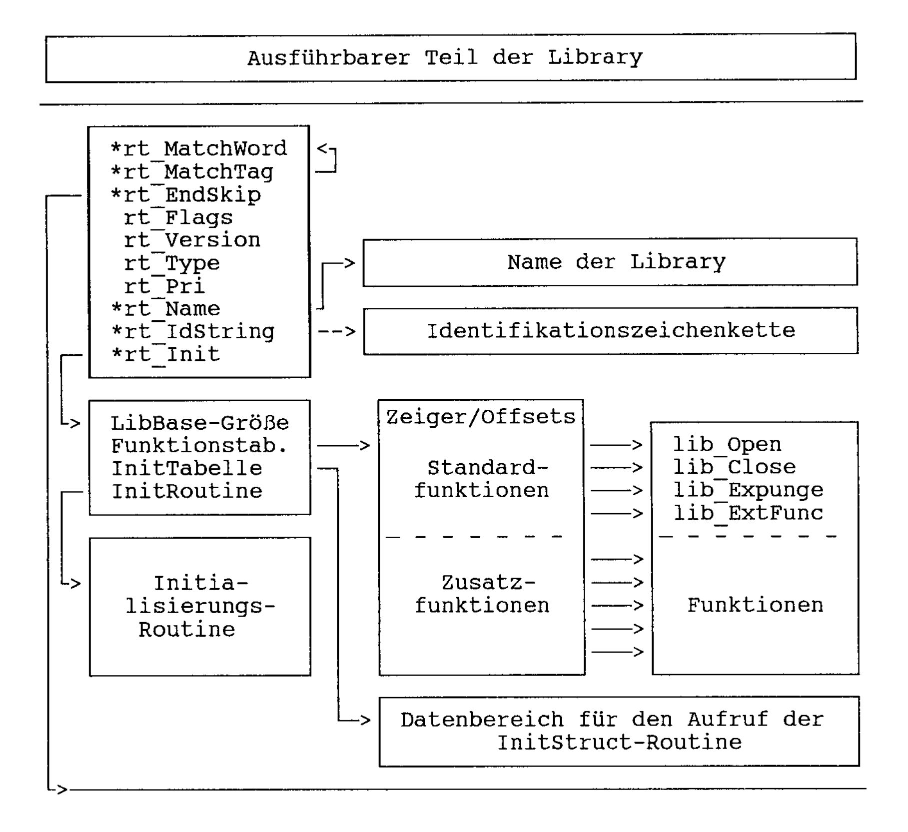
Kompletter Quellcode

Programm zum Ausprobieren

Nachdem wir so viel über den Aufbau der im Speicher befindlichen Libraries und ihre Anwendung gelernt haben, wollen wir uns nun daran machen, eine eigene Library zu konstruieren.

Grundsätzlich ist ein Library ein normales ausführbares Programm. Ein Grund dafür ist, daß die Routinen, nach der Ladung, relokiert werden müssen. Das geht am schnellsten, wenn man sie als ausführbares Programm anlegt.

9.1 Aufbau einer Library-Datei



Dabei besteht das Programm aus zwei Teilen. Zunächst ist da der Programmteil, der am Anfang der Datei stehen muß. Er wird ausgeführt, wenn die Library normal (durch Angabe ihres Namens) aufgerufen wurde. Diese Routine kann z.B. Auskunft

über Version und die Funktionen der Library geben. Zum anderen enthält die Datei eine Resident-Struktur, die mit ihrem Eintrag *rt_EndSkip alle Daten umfaßt, die zur Library gehören. Diese Resident-Struktur wird benutzt, um die Library beim Laden im Speicher richtig zu installieren.

Das Laden und Einrichten verläuft dann wie folgt. Zunächst wird die Library-Datei mit der Funktion LoadSegment in den Speicher geladen und relokiert (dies geht, da es sich um eine ausführbare Datei handelt). Danach wird die geladene Datei nach der Resident-Struktur durchsucht. Ist sie gefunden worden, wird die InitResident-Funktion aufgerufen, um die Library einzubinden. Dabei muß der BCPL-Zeiger, der von der LoadSegment-Funktion zurückgegeben wird, als Parameter übergeben werden.

Wie man sieht, ist das wichtigste an einer Library-Datei die Resident-Struktur. Dabei gibt es zwei Möglichkeiten, die Library mit Hilfe der Resident-Struktur zu initialisieren. Zum einen kann man rt_Init auf eine Routine zeigen lassen, die durch die InitResident-Funktion aufgerufen wird und die Library "von Hand" einrichtet. Zum anderen kann man das Flag AutoInit setzen, welches die InitResident-Funktion veranlaßt, den Eintrag rt_Init als Zeiger auf eine Tabelle mit Registerinhalten für den MakeLibrary-Aufruf zu interpretieren. Nachdem dann die MakeLibrary-Funktion aufgerufen wurde, wird die erstellte Struktur, je nach Typ, in eine der zuständigen System-Listen aufgenommen. Da die zweite Möglichkeit etwas komfortabler ist, haben wir uns bei unserer Library für diese entschieden.

9.2 Die Routine MakeLibrary

An dieser Stelle ist es notwendig, sich die Funktion MakeLibrary, die schon im Kapitel Exec/Library erwähnt wurde, etwas genauer anzusehen. Sie legt eine, durch die angegebenen Parameter definierte Library-Struktur sowie eine Sprungtabelle an.

MakeLibrary = -84

***funcInit (a0)**
Beim ersten Parameter, welcher im Adreßregister 0 übergeben werden muß, handelt es sich um einen Zeiger auf eine Vektorentabelle, die dazu benutzt wird, die Sprungtabelle anzulegen. Dies geschieht mit MakeFunction (siehe Exec/Libraries). Sie bietet zwei Möglichkeiten, die in der Tabelle abgelegten Daten zu verwenden. Entweder kann die Tabelle aus 4 Byte langen Adressen bestehen, welche auf die Routinen zeigen, oder aus 2-Byte-Offsetwerten, die, zur Anfangsadresse der Vektorentabelle addiert, die Adresse der Funktionen ergeben. Beide Tabellen müssen durch eine -1L ($FFFFFFFF) abgeschlos-

sen sein. Zur Erkennung der Offsettabelle muß sie mit dem Wert -1W ($FFFF) beginnen.

Absolute Zeiger:

```
VektorAbsTab:
        dc.l    Funktion_1              ; Adresse Funkt. 1
        dc.l    Funktion_2              ; Adresse Funkt. 2
        dc.l    Funktion_3              ; Adresse Funkt. 3
        dc.l    Funktion_n              ; Adresse Funkt. n
        dc.l    -1                      ; Endmarkierung
```

Offsetwerte:

```
VektorOffTab:
        dc.w    -1                      ; Erkennungsmarke f. Offs.

        dc.w    Funktion_1-VektorOffTab     ; Offset f. Funkt. 1
        dc.w    Funktion_2-VektorOffTab     ; Offset f. Funkt. 2
        dc.w    Funktion_3-VektorOffTab     ; Offset f. Funkt. 3
        dc.w    Funktion_n-VektorOffTab     ; Offset f. Funkt. n
        dc.l    -1                      ; Endmarkierung
```

***structInit (a1)**
Beim zweiten Parameter handelt es sich wiederum um einen Zeiger auf eine Tabelle. Diese Tabelle wird benutzt, um mit Hilfe der Funktion InitStruct eine Library-Struktur anzulegen. Wie die Tabelle aufgebaut werden muß, haben wir im Kapitel Exec gesehen.

***libInit (a2)**
Beim letzten Wert handelt es sich um die Adresse einer Routine, welche von MakeLibrary aufgerufen werden soll, nachdem die Library-Sruktur angelegt worden ist. Ihr werden in den Registern d0 und a0 Werte übergeben, die sie nutzen kann, um private Initialisierungen vorzunehmen. Dabei muß sie den Zeiger auf die Segment-Liste (a0), den die Funktion LoadSegment beim Einladen der Library zurückgegeben hat, zwischenspeichern, um den belegten Speicher der gesamten Library später wieder freigeben zu können. Bei dem Wert im Datenregister d0 handelt es sich um die Basisadresse der Library-Struktur. Der Wert sollte nicht verändert werden, da er direkt als Rückgabeparameter der OpenLibrary-Funktion benutzt wird!

dataSize (d0)
Neben den vorangegangenen Parametern wird auch noch die Länge des Speicherbereichs benötigt, in der die Library-Struktur abgelegt werden soll. Dieser Bereich, der länger sein kann als die eigentliche Library-Struktur, wird zunächst von der MakeLibrary-Funktion gelöscht.

codeSize (d1)
Der letzte Wert, der übergeben werden muß, ist ein Zeiger auf die Segmentliste. Dieser Zeiger wird bei der Parameter-

tabelle einer Resident-Struktur nicht mit übergeben. Erst die OpenLibrary-Funktion, die die Library mittels des LoadSegment-Befehls geladen hat, setzt diesen Parameter.

***libNode (d0)**
Nachdem die Funktion MakeLibrary aufgerufen worden ist, erhält man in d0 die Adresse der Library-Struktur. Zu diesem Zeitpunkt ist sie jedoch noch nicht in eine der System-Listen aufgenommen worden. Dazu dienen die Funktionen AddLibrary, AddDevice oder AddResource. Die InitResident-Funktion erledigt dies auch direkt.

9.3 Die Bestandteile unserer Library

Nach diesem kleinen Exkurs in die Tiefen der MakeLibrary-Funktion sollten wir uns wieder etwas mehr um unsere eigene Library kümmern.

Anfangen wollen wir mit der Resident-Struktur.

```
LibResident:
        dc.w    $4AFC                   ;┐ dient zur Erkennung der
        dc.l    LibResident             ;┘ Resident-Struktur
        dc.l    EndResident             ; Ende der Resident-Daten
        dc.b    %10000000               ; AutoInit-Flag setzen
        dc.b    1                       ; Version
        dc.b    9                       ; Typ = nt_Library
        dc.b    0                       ; Priorität soll Null sein
        dc.l    LibName                 ; Zeiger auf Librarynamen
        dc.l    LibIDString             ; Zeiger auf ID-String
        dc.l    LibInitTab              ; Zeiger auf Tabelle mit den
                                        ; Parametern für den
                                        ; MakeLibrary-Aufruf

LibName:  dc.b    "test.library",0
          even
LibIDString:      dc.b  "Test-Library (Ron) v0.1",0
          even
```

Bild 9.1: Resident-Struktur der Library

Die Bedeutung der einzelnen Einträge der Resident-Struktur dürfte eigentlich kein Problem mehr darstellen. Deshalb sparen wir uns weitere Erklärungen. Jedoch müssen wir auf die Parameter-Tabelle für den MakeLibrary-Aufruf etwas näher eingehen.

```
LibInitTab:
        dc.l    42                      ; LibSize (34) +
                                        ; SegList (4)  +
```

```
                                        ; DosBase (4)  = 42
        dc.l    FuncTab                 ; Tabelle der Vektoren
        dc.l    DataTab                 ; Initialsierungstabelle
        dc.l    LibInitRout             ; Initialisierungroutine
```

Bild 9.2: Parametertabelle für MakeLibrary

Der erste Wert der LibInitTabelle gibt die Größe der Basis-Struktur der Library an. Hierbei muß beachtet werden, daß man die Länge der eigentlichen Library-Struktur nicht vergißt. Für unsere Library wollen wir noch zwei weitere Werte ablegen. Zum einen die Adresse der Segment-Liste, die unserer Initialisierungs-Routine übergeben worden ist (LibInitRout). Zum anderen den Zeiger auf die Dos-Library, deren Funktionen von unserer Routine benutzt werden. Daraus ergibt sich eine Länge von 42 Byte für die Basis-Struktur unserer Library.

Als nächstes wird ein Zeiger auf die Vektorentabelle erwartet, deren Aufbau wir schon beschrieben haben.

```
FuncTab:
        dc.l    Open                    ; -6  ┐
        dc.l    Close                   ; -12 │ Standardfunktionen
        dc.l    Expunge                 ; -18 │ der Library
        dc.l    ExtFunc                 ; -24 ┘

        dc.l    WriteLn                 ; -30   Erste Spezialfunktion

        dc.l    -1                      ; Endkennung
```

Bild 9.3: Vektorentabelle der Library

Neben den vier Standardfunktionen, die jede Library benötigt, bauen wir nur eine weitere Funktion ein (WriteLn - TurboPascal läßt grüßen !!!). Sie soll lediglich eine mit einem Null-Byte beendete Zeichenkette ausgeben. Dies sollte zur Demonstration für unsere Library vorerst ausreichen.

Nun aber wieder zurück zur LibInit-Tabelle. Der nächste Eintrag ist der Zeiger auf die Daten-Tabelle, die für das Anlegen der Library-Struktur verwendet wird.

```
DataTab:
        dc.b    %11100000,0             ; 24-Bit Offset-Befehl
        dc.w    8                       ; Offset
        dc.b    9,0                     ; Datenwert (Typ des Knotens)

        dc.b    %11000000,0             ; 24-Bit Offset-Befehl
        dc.w    10                      ; Offset
        dc.l    LibName                 ; Datenwert (Name der Lib)
```

```
dc.b    %11100000,0          ; 24-Bit Offset-Befehl
dc.w    14                   ; Offset
dc.b    6,0                  ; Datenwert (Library-Flags)

dc.b    %11010001,0          ; 24-Bit Offset-Befehl
dc.w    20                   ; Offset
dc.w    3,4                  ; Datenwert (Version, Rever.)

dc.l    0                    ; Endkennung
```

Bild 9.4: Datentabelle für LibInit-Tabelle

Auch hier gehen wir nicht auf den Aufbau der Tabelle ein, da sie im Kapitel Exec schon ausführlich beschrieben wurde. Jedoch muß man darauf achten, daß man nicht die Einträge lib_NegSize und lib_PosSize überschreibt, da sie zuvor schon initialisiert worden sind. Zur Kontrolle folgt nun die Struktur, die durch die Initialisierungs-Tabelle erstellt werden soll.

TestLib-Struktur:

```
00    ds.l    1          ;┐             Platz für lib_Succ
04    ds.l    1          ;│  Node-           und lib_Pred
08    dc.b    9          ;│ Struktur    lib_Type
09    dc.b    0          ;│             lib_Pri
10    dc.l    LibName    ;┘             lib_Name
                       ;|
14    dc.b    6          ;│             lib_Flags
15    dc.b    0          ;│             lib_Pad
16    ds.w    1          ;│ Library-    Platz für lib_NegSize
18    ds.w    1          ;│ Struktur         und lib_PosSize
20    dc.w    3          ;│             lib_Version
22    dc.w    4          ;│             lib_Reversion
24    ds.l    1          ;│             Platz für lib_IDString
28    ds.l    1          ;│                       lib_Sum
32    ds.w    1          ;┘                  und lib_OpenCnt

38    ds.l    1          ;┐ private  Zeiger auf Segment-Liste
42    ds.l    1          ;┘  Daten   Basisadresse der DosLib.

; Alle Werte, die nur Speicher belegen (ds.x x) und keinen
; bestimmten Wert haben (dc.x x), benötigen zur Zeit nur den
; vordefinierten Wert Null.
```

Bild 9.5: Teststruktur für Library

Nach den Initialisierungsdaten bleibt jetzt nur noch der Zeiger auf die Initialisierungs-Routine, welche von MakeLibrary aufgerufen wird. Dabei erhält man als Parameter in d0 den Zeiger auf die Library-Struktur und in a0 die Adresse der Segment-Liste.

Zunächst retten wir die Adreßregister 4-6 auf den Stack und legen dann die Basisadresse unserer Library in a5 ab.

```
        move.l  a4-a6,-(a7)
        move.l  d0,a5
```

Nun können wir mit dem Offsetwert 34 über das Adreßregister 5 die Adresse der Segment-Liste in unsere Library-Struktur eintragen.

```
        move.l  a0,34(a5)
```

Jetzt öffnen wir die Dos-Library und legen den Zeiger auf die Basis-Struktur ebenfalls in unserer Library-Struktur ab. Sollte ein Fehler auftreten, so wird die Routine umgehend verlassen und in d0 eine Null übergeben.

```
        move.l  ExecBase,a6     ; Exec-Base nach a6
        moveq   #0,d0           ; Version ist bedeutungslos
        lea     DosName(pc),a1  ; Zeiger auf Zeichenkette
        jsr     OpenLib(a6)     ; Library öffnen
        move.l  d0,38(a5)       ; DosBase eintragen

        beq     LibInitEnd      ; Fehler aufgetreten ?
```

Zum Schluß müssen wir die Basisadresse unserer Library nach d0 kopieren, da dieser Wert nachher zurückgegeben werden soll. Außerdem müssen wir noch die Register (a4-a6) restaurieren.

```
        move.l    a5,d0               ; Lirary-Basis übergeben

LibInitEnd:
        movem.l   (a7)+,a4-a6         ; Register restaurieren
        rts
```

Nachdem wir uns um die Initialisierung der Library gekümmert haben, sollten wir uns die Standard-Funktionen ansehen. Wie wir schon im Library-Kapitel beschrieben haben, sind die ersten vier Offsetwerte (-6, -12, -18, -24) durch Standardfunktionen belegt. Dabei werden die Funktionen von den System-Routinen OpenLibrary, CloseLibrary und RemLibrary aufgerufen, um der Library die Chance zu geben, auf das Öffnen, Schließen und Entfernen selbst zu reagieren.

Open-Funktion

Die Open-Funktion hat nur einen kleinen Aufgabenbereich. Sie muß das DelayExpunge-Bit im Flag-Eintrag der Library-Struktur löschen, damit die Library nicht entfernt wird. Außerdem erhöht sie den OpenCnt-Wert, der die Anzahl der zugreifenden Tasks festhält. Schließlich muß sie noch den Zeiger auf die Library-Struktur ins Datenregister 0 kopieren.

```
; Die Open-Funktion der Library erhält von der OpenLibrary-
; Funktion von Exec zwei Parameter. Zum einen die Version
; (d0) und zum anderen den Zeiger auf die Library-
; Struktur (a6)

Open:
        bclr     #3,14(a6)         ; Expunge-Bit löschen
        addq.w   #1,32(a6)         ; OpenCnt erhöhen
        move.l   a6,d0             ; Zeiger auf Basis in d0
                                   ; übergeben.
        rts                        ; Fertig !
```

Close-Funktion

Auch die Close-Funktion muß verhältnismäßig wenig erledigen. Zunächst wird der OpenCnt-Wert erniedrigt. Sollte kein Benutzer mehr vorhanden sein, wird kontrolliert, ob die Library entfernt werden darf (DelayExpunge-Bit = 1). Ist dies der Fall, wird sie mit Hilfe der Expunge-Funktion entfernt. Wichtig ist dabei, daß das Datenregister 0 beim Verlassen der Funktion unbedingt mit Null initialisiert ist. Die CloseLibrary-Funktion, die "uns" gerufen hat, würde sonst den Wert als Zeiger für die Segment-Liste interpretieren und den Speicher mittels UnLoadSeg freigeben.

```
; Die Close-Funktion erhält Parameter von der CloseLibrary-
; Funktion. Es handelt sich hierbei um die Basisadresse der
; Library, die in a6 übergeben wird.

Close:
        moveq    #0,d0             ; d0 löschen, da hier ein
                                   ; Zeiger auf die Segment-
                                   ; Liste erwartet wird.

        subq.w   #1,32(a6)         ; OpenCnt erniedrigen
        bne      CloseEnd          ; Wert ist > Null

        btst     #3,14(a6)         ; darf die Library entfernt
        beq      CloseEnd          ; werden ?

        bsr      Expunge           ; Ja, dann entfernen

CloseEnd:
        rts                        ; Ende
```

Expunge-Funktion

Die Expunge-Funktion (von RemLibrary aufgerufen) fällt etwas länger aus, da sie die Library aus dem System entfernen muß. Dabei werden auch hier zunächst die Register gerettet (d2, a5, a6). Dann wird kontrolliert, ob die Library von einem Benutzer noch benötigt wird (OpenCnt>=0 ?). Sollte sie jetzt noch nicht entfernt werden können, wird das Expunge-Bit gesetzt. Dadurch wird angemeldet, daß die Library entfernt

werden soll. Schließt nun der letzte Benutzer die Library, so wird automatisch aus der Close-Funktion die Expunge-Funktion aufgerufen, welche die Library endgültig aus dem System "wirft". Diese Methode erinnert ein bißchen an: "Der letzte macht die Tür zu!".

```
; Die Expunge-Funktion benötigt in a6 die Basisadresse.

Expunge:
        movem.l  d1-d2/a5-a6,-(a7)      ; Register retten

        tst.w    32(a6)          ; kein Benutzer mehr ?
        beq      ExpungeBranch   ; doch, dann verzweigen
```

Zuerst wird kontrolliert, ob die Library noch benutzt wird. Sollte das der Fall sein, so wird das Delay-Expunge-Bit ("verzögerte Entfernung") gesetzt, um die Library bei nächster Gelegenheit zu entfernen. Außerdem wird das Datenregister 0 gelöscht, in dem sonst ein Zeiger auf die Segment-Liste erwartet wird. Dann wird die Expunge-Routine verlassen.

```
        moveq    #0,d0           ; Segment-List = 0
        bset     #3,14(a6)       ; Delay-Expunge-Bit setzen
        bra      ExpungeEnd      ; jetzt noch nicht entfernen
```

Soll die Library wirklich entfernt werden, so wird zunächst die Dos-Library geschlossen, die wir ja beim Initialisieren geöffnet haben.

```
ExpungeBranch:
        move.l   a6,a4           ; Zeiger auf Base retten
        move.l   38(a4),a1       ; Adresse der DosBase lesen
        move.l   ExecBase,a6     ; um sie anschließend zu
        jsr      CloseLib(a6)    ; schließen
```

Jetzt wird die Library aus der Library-Liste gelöscht, und der Zeiger auf die Segment-Liste ins Datenregister 2 gerettet.

```
        move.l   a4,a1           ; Zeiger auf Library-Node
        jsr      Remove(a6)      ; Remove
        move.l   34(a4),d2       ; SegmentList retten
```

Zwar wird der Speicher, den die Funktionen belegen, von Exec durch UnLoadSeg freigegeben, jedoch muß er für die Sprungtabelle und die Library-Struktur "von Hand" befreit werden. Hierzu berechnet man die Anfangsadresse (Library-Base - lib_NegSize) und die Länge (lib_NegSize + lib_PosSize) mit Hilfe der abgelegten Werte.

```
        moveq    #0,d0           ; d0 löschen
        move.w   16(a4),d0       ; lib_NegSize auslesen
        move.l   a4,a1           ; Basisadresse nach a1
        sub.l    d0,a1           ; lib_NegSize abziehen
        add.w    18(a4),d0       ; lib_PosSize + lib_NegSize
        jsr      FreeMem(a6)     ; FreeMem
```

Zum Schluß muß der gerettete Zeiger auf die Segment-Liste in d0 kopiert werden, da er von Exec benötigt wird, um den Speicher, den die Library belegt hat, mit der UnLoadSeg-Funktion freizugeben. Dann werden die Register wieder restauriert und die Expunge-Routine verlassen.

```
        move.l  d2,d0           ; SegmentList zurückgeben

ExpungeEnd:
        movem.l (a7)+,d1-d2/a5-a6     ; Register restaurieren
        rts
```

ExtFunc-Funktion

Die ExtFunc-Funktion wird zur Zeit nicht unterstützt. Man sollte trotzdem eine kleine Routine einbinden, die das Datenregister 0 löscht und dann zurückkehrt.

```
ExtFunc:
        moveq   #0,d0           ; Datenregister 0 löschen
        rts                     ; Funktion beenden
```

Dies waren alle wichtigen Teile, aus denen eine Library besteht. Natürlich fehlen jetzt noch die Routinen, die verwaltet werden sollen. Zu Demonstrationszwecken haben wir eine kleine Funktion eingebunden, die eine mit Null beendete Zeichenkette in den Standardausgabekanal (meist ist damit das CLI-Fenster gemeint) ausgibt.

WriteLn	**=**	**-30**

***String**	**a0**	**<**	Zeiger auf eine mit Null abgeschlossene Zeichenkette.
Erklärung			Durch die Funktion WriteLn wird eine mit Null abgeschlossene Zeichenkette ausgegeben.

9.4 Der Quellcode der Library

Wir haben zwar den Aufbau der Library detailliert beschrieben, uns aber dennoch dazu entschlossen, das vollständige Listing abzudrucken; denn anhand des zusammenhängenden Listings kann man den Aufbau wesentlich leichter verstehen.

```
*
* Kapitel 9
* Quelltext der "test.library"
*

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
Output          =       -60
Write           =       -48
Remove          =       -252
FreeMem         =       -210

; Zuerst kommt der "CLI-Teil".

Start:
        move.l  ExecBase,a6     ; Dos-Library öffnen
        lea     DosName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)
        move.l  d0,a6
        beq     DosError

        jsr     Output(a6)      ; Standardausgabekanal
        move.l  d0,d1           ; ermitteln

        move.l  #TextA,d2       ; Adresse der Zeichenkette
        move.l  #TextE-TextA,d3         ; und Länge übergeben.
        jsr     Write(a6)       ; Text ausgeben

        move.l  a6,a1           ; Dos-Library wieder
        move.l  ExecBase,a6     ; schließen
        jsr     CloseLib(a6)

DosError:                       ; Fehler-Nummer Null
        moveq  #0,d0            ; übergeben
        rts

TextA:          dc.b    10,"> TestLibrary Version 0.1 (Ron) <",10,10
TextE:
                even

; Anfang der eigentlichen Library

LibResident:
                dc.w    $4AFC           ; (Illegal)
                dc.l    LibResident     ; rt_MatchTag
                dc.l    EndResident     ; rt_EndSkip
                dc.b    %10000000       ; rt_Flags
                dc.b    1               ; rt_Version
                dc.b    9               ; rt_Type
                dc.b    0               ; rt_Pri
                dc.l    LibName         ; rt_Name
                dc.l    LibIDString     ; rt_IDString
```

```
            dc.l    LibInitData    ; rt_Init

LibName:    dc.b    "test.library",0     ; Library-Name
            even
LibIDString:dc.b    "Test-Library v0.1 (RON)",0
            even

LibInitData:
            dc.l    42             ; LibSize (34)
                                   ; SegList (4)
                                   ; DosBase (4) = 42
            dc.l    FuncTab
            dc.l    DataTab
            dc.l    LibInitRout

FuncTab:
            dc.l    Open           ;┐
            dc.l    Close          ;│ Standardfunktionen
            dc.l    Expunge        ;│
            dc.l    ExtFunc        ;┘

            dc.l    WriteLn        ; Spezialfunktion

            dc.l    -1             ; Endkennung

DataTab:
            dc.b    %11100000,0
            dc.w    8              ;
            dc.b    9,0            ; lib_Typ

            dc.b    %11000000,0
            dc.w    10
            dc.l    LibName        ; lib_Name

            dc.b    %11100000,0
            dc.w    14
            dc.b    6,0            ; lib_Flags

            dc.b    %11000000,0
            dc.w    20
            dc.w    3,4            ; lib_Version + lib_Reversion

            dc.l    0              ; Endkennung

DosName:    dc.b "dos.library",0
            even


; Nun folgt die Initialisierungsroutine

LibInitRout:
        movem.l  a4-a6,-(a7)     ; Register retten

        move.l   d0,a5           ; Library-Basis nach a4

        move.l   a0,34(a5)       ; Segment-Liste eintragen
```

```
        move.l   ExecBase,a6      ; DosLibrary öffnen
        moveq    #0,d0
        lea      DosName(pc),a1
        jsr      OpenLib(a6)
        move.l   d0,38(a5)        ; DosBase eintragen
        beq      LibInitEnd

        move.l   a5,d0            ; Liary-Basis übergeben
LibInitEnd:
        movem.l  (a7)+,a4-a6      ; Register restaurieren
        rts

; Die Open-Funktion wird von der OpenLibrary-Routine
; aufgerufen

Open:
        bclr     #3,14(a6)        ; ExpungeBit löschen
        addq.w   #1,32(a6)        ; OpenCnt erhöhen
        move.l   a6,d0            ; Basisadresse übergeben
        rts

; Die Close-Funktion wird von der CloseLibrary-Funktion
; aufgerufen

Close:
        moveq    #0,d0            ; SegmentList-Ptr = 0

        subq.w   #1,32(a6)        ; OpenCnt verringern
        bne      CloseEnd         ; noch nich Null, dann Ende

        btst     #3,14(a6)        ; ExpungeBit gesetzt ?
        beq      CloseEnd         ; Nein, dann nicht entfernen

        bsr      Expunge          ; Library endgültig entfernen

CloseEnd:
        rts

; Nun folgt die Expunge-Routine, die die Library aus dem
; System entfernt. Sie wird von der RemLibrary-Routine und
; der Close-Funktion aufgerufen.

Expunge:
        movem.l  d1-d2/a5-a6,-(a7)       ; Register retten

        tst.w    32(a6)           ; sind noch Benutzer
        beq      ExpungeBranch    ; vorhanden? Nein, dann
                                  ; Library entfernen

        moveq    #0,d0            ; SegmentList-Ptr = 0
        bset     #3,14(a6)        ; ExpungeBit setzen, damit
        bra      ExpungeEnd       ; die Library beim nächsten
                                  ; mal entfernt werden kann
ExpungeBranch:
        move.l   a6,a4            ; DosLibrary schließen
```

```
        move.l  38(a4),a1
        move.l  ExecBase,a6
        jsr     CloseLib(a6)
        move.l  a4,a1           ; Library aus Lib-List nehmen
        jsr     Remove(a6)      ; Remove

        move.l  34(a4),d2       ; SegmentList-Ptr retten

        moveq   #0,d0           ; Speicher der Library-
        move.w  16(a4),d0       ; Struktur und der Sprung-
        move.l  a4,a1           ; tabelle freigeben
        sub.l   d0,a1
        add.w   18(a4),d0
        jsr     FreeMem(a6)     ; FreeMem

        move.l  d2,d0           ; Zeiger auf Segment-Liste
                                ; zurückgeben
ExpungeEnd:
        movem.l (a7)+,d1-d2/a5-a6     ; Register restaurieren
        rts

; Die letzte Standardfunktion wird zur Zeit nicht benutzt.

ExtFunc:
        moveq   #0,d0           ; d0 löschen und zurück
        rts

; Nun kommt unsere erste eigene Funktion. Sie erwartet in
; a0 einen Zeiger auf eine Zeichenkette, die ausgegeben
; werden soll.

WriteLn:
        movem.l a0/a6,-(a7)     ; Zeiger und Base retten

        move.l  a6,a5           ; DosLibrary-Adresse
        move.l  38(a6),a6       ; auslesen

        jsr     Output(a6)      ; Standardausgabekanal
        move.l  d0,d1           ; ermitteln

        move.l  (a7)+,a0        ; a0 restaurieren
        move.l  a0,d2           ; Adresse der Zeichenkette
        move.l  #-1,d3          ; nach d2 kopieren. Nun wird
WLLoop:                         ; die Länge der Zeichenkette
        addq.l  #1,d3           ; ertestet.
        tst.b   (a0)+
        bne     WLLoop

        jsr     Write(a6)       ; Zeichenkette ausgeben
        move.l  (a7)+,a6        ; BasePtr restaurieren
        rts

EndResident:
```

Programm "test.library.s" (Quelltext der "Test-Library")

9.5 Programm zum "Ausprobieren"

Um nun unsere Library auszuprobieren, müssen wir sie zunächst assemblieren und unter dem Namen "test.library" im Verzeichnis "libs:" ablegen. Dann können wir sie, wie jede andere Library auch, öffnen und ihre Funktion (WriteLn) benutzen.

```
* Kapitel 9
* Demonstrationsprogramm für die "test.library"
*

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
WriteLn         =       -30 ; Offset unserer Funktion

        move.l  ExecBase,a6     ; Basisadresse der ExecLib
        move.l  #TestName,a1    ; Zeiger auf Zeichenkette
        moveq   #0,d0           ; Version ist belanglos
        jsr     OpenLib(a6)     ; Library öffnen
        move.l  d0,TestBase     ; Basisadresse speichern
        beq     Error           ; Fehler aufgetreten ?

        move.l  d0,a6           ; Basisadresse nach a6
        lea     Text,a0         ; Text als Parameter nach a0
        jsr     WriteLn(a6)     ; WriteLn aufrufen

        move.l  ExecBase,a6     ; ExecBase
        move.l  TestBase,a1     ; Zeiger auf unsere Library

; Wenn wir möchten, daß die Library aus dem Speicher
; entfernt werden soll, können wir das zum einen mit der
; RemLibrary-Funktion erledigen, oder wir setzen einfach das
; Expunge-Bit von "Hand". Dann wird nämlich die Expunge-
; Funktion von der Close-Funktion, die wiederum von der
; CloseLibrary-Funktion angesprungen wurde, aufgerufen.

        bset    #3,14(a1)       ; Expunge-Bit setzen

        jsr     CloseLib(a6)    ; Lib schließen und entfernen

Error:
        rts                     ; Progamm beenden

* Datenbereich

Text:           dc.b 10,"Dies ist ein <Hello world>-Text",10,0
                even
```

```
TestName:      dc.b    "test.library",0    ; Name unserer Lib.
               even

TestBase:      dc.l    0                   ; Var. für Basisadr.
```

Programm 9.1: Testprogramm für eigene Library

Kapitel 10
Die Devices

Die (selbstgeschriebene) IO-Library

Übersicht über die Devices

Das Trackdisk-Device

Das Printer-Device

Das Console-Device

Das Narrator-Device

10.1 Die IO-Library

Bevor wir uns nun auf die einzelnen Devices "stürzen", möchten wir noch auf unsere selbstgeschriebene "io.library" hinweisen. Diese Library enthält sechs Funktionen, die uns die Arbeit mit Devices erleichtern sollen. So kann man z.B. mit einem Funktionsaufruf eine gesamte IORequest-Struktur initialisieren lassen. Dies umschließt das Anlegen eines Message-(Reply-)Ports, das Anlegen einer IORequest-Struktur und das Öffnen des angegebenen Devices. Dadurch sind die folgenden Demonstrationsprogramme für die einzelnen Devices wesentlich kompakter geworden.

Damit man die Funktionen der Library auch in eigenen Programmen benutzen kann, sollen nun die Erklärungen der Funktionen folgen. Aus Platzgründen haben wir den Quelltext der Library nicht abgedruckt. Jedoch befindet sich auf der beiliegenden Diskette die Library sowohl in assemblierter Form wie auch als Quelltext.

CreatePort		=	**-30 (IO-Library)**
***PortName**	**a0**	<	Zeiger auf Zeichenkette mit dem Namen des Message-Ports. Wird a0 mit einer Null initialisiert, wird dem Port kein Name gegeben. Außerdem wird er dann nicht in die PublicPort-Liste aufgenommen.
Pri	**d0**	<	Priorität des Ports.
***Port**	**d0**	>	Zeiger auf die angelegte Message-Port-Struktur oder eine Null, wenn ein Fehler aufgetreten ist.
Erklärung			Es wird eine MsgPort-Struktur angelegt, die, wenn ein Zeiger auf einen Namen angegeben worden ist, in die PublicPort-Liste aufgenommen wird. Sollte es einen Port mit dem angegebenen Namen schon in der PublicPort-Liste geben, so wird kein MsgPort angelegt.

DeletePort		=	**-36 (IO-Library)**
***Port**	**a0**	<	Zeiger auf eine von CreatePort angelegte Message-Port-Struktur.
Erklärung			Gibt den durch die angegebene MsgPort-Struktur belegten Speicher wieder frei und entfernt den Port gegebenenfalls aus der PublicPort-Liste.

CreateIOReq	=	-42 (IO-Library)

*ReplyPort	a0	<	Zeiger auf einen MsgPort, der als ReplyPort in die IORequest-Struktur eingetragen werden soll.
Size	d0	<	Größe der anzulegenden IORequest-Struktur.
*IOReq	d0	>	Zeiger auf die angelegte und initialisierte IORequest-Struktur oder eine Null, wenn ein Fehler aufgetreten ist.
Erklärung			Legt eine IORequest-Struktur mit der angegebenen Größe an und richtet sie ein.

DeleteIOReq	=	-48 (IO-Library)

*IOReq	a0	<	Zeiger auf eine mit CreateIOReq angelegte IORequest-Struktur.
Erklärung			Gibt den von der angegebenen IORequest-Struktur belegten Speicher wieder frei.

CreateIOFast	=	-54 (IO-Library)

*DevName	a0	<	Zeiger auf Namen des Devices, welches geöffnet werden soll.
UnitNum	d0	<	Unit-Nummer, die beim Öffnen des Devices übergeben werden soll.
Flags	d1	<	Flag-Wert, der beim Öffnen des Devices übergeben werden soll.
Size	d2	<	Größe der IORequest-Struktur, die angelegt werden soll.
*IOReq	d0	>	Zeiger auf angelegte IORequest-Struktur oder eine Null, wenn ein Fehler aufgetreten ist.
Erklärung			Diese Funktion richtet zunächst einen Message-Port ohne Namen ein, dessen Adresse in die danach angelegte IORequest-Struktur eingetragen wird. Zum Schluß wird das angegebene Device geöffnet. Man erhält als Rückgabewert einen Zeiger auf die IORequest-Struktur, mit der man sofort mit dem Device kommunizieren kann.

DeleteIOFast	=	**-60 (IO-Library)**

***IOReq**	**a0**	<	Zeiger auf IORequest-Struktur, die mittels der Funktion CreateIOFast angelegt worden ist.
Erklärung			Schließt das geöffnete Device und gibt den belegten Speicher wieder frei.

Auf ein Demoprogramm kann hier wohl verzichtet werden, da die IO-Library in den folgenden Programmen des öfteren Verwendung finden wird.

Noch ein Hinweis: Da das System Libraries, die geöffnet werden sollen, immer im LIBS:-Verzeichnis, d.h. im Libs-Verzeichnis der Startdiskette, sucht, müssen Sie die selbstgeschriebenen Libraries von der Buch-Programmdiskette auf die Diskette, die Sie zum Systemstart verwenden wollen, kopieren. Das selbe gilt auch für die Devices (siehe Kapitel 11).

10.2 Kurzübersicht über die Devices

Im Device-Teil des Exec-Kapitel haben wir uns schon mit dem allgemeinen Aufbau eines Devices und mit den zur Benutzung notwendigen Schritten befaßt. In diesem Kapitel wollen wir nun auf die Funktionen der einzelnen Devices etwas intensiver eingehen. Bevor wir damit anfangen, hier ein kurzer Überblick über die wichtigsten Devices und ihre Aufgabengebiete:

audio.device	Steuert die Audio-Hardware des Amiga. Das Audio-Device ist zuständig für das Abspielen von Sound-Samples.
console.device	Das Console-Device ist quasi die Exec-Instanz eines DOS-Fensters (mit dem Dateinamen "CON:..." geöffnet). Es tätigt Ein- und Ausgaben in solchen Fenstern.
gameport.device	Dieses Device dient zur Abfrage der Geräte am Gameport (Maus und Joystick).
input.device	Hier laufen alle Eingabe-Ereignisse (Tastendrücke, Mausklicks etc.) zusammen und werden an die Intuition-Library weitergeleitet.
keyboard.device	Ist zuständig für die Umrechnung der Raw-Key-Codes ('rohe' Tastencodes, wie sie vom Tastaturprozessor kommen) in ASCII-Codes gemäß der eingestellten Keymap.
narrator.device	Das "Quassel-Gerät" (Sprachausgabe). Die Texte, die ausgegeben werden sollen, müssen als Phonem-Codes (Lautschrift) vorliegen. Eine entspre-

	chende Umwandlung von Klartexten nimmt die "translator.library" vor.
parallel.device	Ist zuständig für die Datenübertragung über den Parallel-Port.
printer.device	Setzt auszugebende Texte in Druckerkommandos um und bedient das Parallel- oder Serial-Device (je nachdem, an welchem Port Ihr Drucker angeschlossen ist).
serial.device	Datenübertragung über der seriellen Port. Haupteinsatzgebiet sind hier Modems, aber auch manche Drucker werden am seriellen Port angeschlossen.
trackdisk.device	Steuert die Floppy-Hardware des Amiga. Das Trackdisk-Device kann Daten auf Disketten schreiben, sie lesen, Disketten formatieren usw.

10.3 Das Trackdisk-Device

Als erstes Device wollen wir uns nun das Trackdisk-Device (abgekürzt TDD) unter die Lupe nehmen. Es ist für die Verwaltung der Diskettenlaufwerke zuständig. Seine Hauptaufgaben sind das Lesen und Schreiben von Daten und die Formatierung von Disketten. Es bietet aber auch noch andere Möglichkeiten, z.B. die Positionierung des Schreib/Lese-Kopfes, die Abfrage, ob eine Disk im Laufwerk liegt und ob sie schreibgeschützt ist, und die automatische Erkennung von Diskwechselvorgängen.

Bevor wir aber zum Aufbau und zur Funktionsweise dieses Gerätes kommen, eignen wir uns zunächst etwas Hintergrundwissen an, indem wir fragen: Wie ist eine Diskette überhaupt aufgebaut?

10.3.1 Der physikalische Aufbau einer Diskette

Wie Sie sicherlich wissen, besteht das "Innere" einer Diskette aus einer runden, magnetisierbaren Scheibe (auch wenn sie von außen eckig aussieht). Auf dieser Scheibe werden Daten aufgezeichnet, indem sie vom Schreib/Lese-Kopf an bestimmten Stellen mit einer bestimmten Magnetisierung versehen wird. Wie das exakt funktioniert, ist Sache der Hardware-Programmierer, die kein Trackdisk-Device benutzen können oder wollen und die Laufwerke auf der untersten Ebene, über die Hardware-Register, ansprechen. Das ist ein äußerst kompliziertes Unterfangen, aber wir haben ja zum Glück das TDD, das uns in dieser Hinsicht viel Ärger ersparen wird. Wir werden auf diese Art der Floppyprogrammierung kurz im Zusammenhang mit zwei TDD-Kommandos zu sprechen kommen.

Ansonsten ist für uns nur wichtig zu wissen, wie wir das TDD dazu bringen, unsere Daten zu speichern.

Der Amiga teilt eine Diskette in 80 konzentrische Ringe, die sog. "Spuren" oder "Tracks" ein. Diese Tracks werden durchnummeriert, wobei die Spur mit der niedrigsten Nummer (0) ganz außen und die mit der höchsten (79) ganz innen liegt. Jeder Track wird nun, wie ein Kuchenstück, in 11 "Sektoren" eingeteilt. Damit haben wir 80*11 = 880 Sektoren auf einer Diskseite. Nun hat eine Diskette aber bekanntlich zwei Seiten, die auch beide vom Laufwerk benutzt werden. Es hat zwei Schreib/Lese-Köpfe, je einen für die Ober- und Unterseite. Den unteren bezeichnet man auch als "Head 0" und den oberen als "Head 1". Eine Spur mit ihren beiden Seiten zusammengenommen nennt man "Zylinder".

Eine Kombination aus Spur, Sektor und Kopf nennt sich "Block". Fix gerechnet hat eine Amiga-Diskette also 80*11*2 = 1760 Blöcke. Man kann einen Block also eindeutig über seine Blocknummer (0-1759) oder über die Spur-Kopf-Sektor-Nummer identifizieren.

Manchmal wird die Kopfnummer auch nicht extra angegeben, sondern in der Spurnummer untergebracht. Jeder Zylinder bekommt dann zwei Spurnummern, je eine für Ober- und Unterseite. Spur 0 ist demnach Zylinder 0, Seite 0, Spur 1 ist Zylinder 0, Seite 1, Spur 2 ist Zylinder 1, Seite 0 usw.

Jeder Block hat eine Größe von 512 Bytes. Eine Amiga-Diskette hat also einen Speicherplatz von 1760*512 = 901120 Bytes (oder 880 KBytes). Eigentlich eine ganze Menge auf einer so kleinen Scheibe.

Die Einteilung in Spuren und Sektoren gilt natürlich nicht nur für Disketten, sondern auch für Festplatten und andere vergleichbare Speichermedien. Der Unterschied ist nur, daß die Spur- und Sektoranzahl bei Festplatten in einer ganz anderen Größenordnungen liegt. Heutzutage sind Festplatten, deren Magnetscheibe die Größe einer Diskette hat, schon mit Speicherkapazitäten von mehr als 100 MByte (1 MByte = 1024 KByte) zu haben.

10.3.2 Öffnen des TDD und IORequest-Struktur

Geöffnet wird das Trackdisk-Device ganz normal über die OpenDevice-Routine von Exec. Als Unit-Nummer muß die Nummer des gewünschten Laufwerks (0-3) angegeben werden. Als Flag ist hier TD_ALLOW_NON_3_5 möglich. Dieses Flag bewirkt, daß das Trackdisk-Device auch Laufwerke annimmt, die eine andere Größe als 3 1/2 Zoll haben. Ist das Flag nicht gesetzt, werden nur 3 1/2 Zoll-Laufwerke akzeptiert.

Falls beim Öffnen ein Fehler auftrat, enthält d0 nach dem Aufruf einen Wert <> 0, ansonsten steht es auf 0. IORequest-Zugriffe mittels der so initialisierten IOStdReq-Struktur beziehen sich immer nur auf die Laufwerksunit, die beim Öffnen des Devices angegeben wurde!

Das Trackdisk-Device verwendet im Normalfalle für seine Kommandos die IOStdReq-Struktur. Im Exec-Kapitel haben wir diese Struktur schon kennengelernt. Weiterhin kennt das TDD einige sog. "erweiterte" Kommandos, die eine ebensolche Struktur (IOExtTD - Erweiterter Trackdisk-IO) benötigen. Da wir aber wie immer vom Einfachen zum Komplizierten gehen (dabei ist die IOExtTD-Struktur gar nicht kompliziert - sie umfaßt nur zwei Einträge), beschäftigen wir uns zunächst mit den "normalen" Befehlen und dem IOStdReq.

Die IOStdReq-Struktur (für Trackdisk-Device)

```
00   ds.b    io_Message,20        ; Zugehörige Message-Struktur
20   dc.l   *io_Device            ; Zeiger auf Device
24   dc.l   *io_Unit              ; Zeiger auf Unit
28   dc.w    io_Command           ; Trackdisk-Kommando
30   dc.b    io_Flags             ; Flags
31   dc.b    io_Error             ; Eventueller Fehler
32   dc.l    io_Actual            ; Diverse Rückgabewerte
36   dc.l    io_Length            ; Anzahl der zu übertragenden Bytes
40   dc.l   *io_Data              ; Zeiger auf Daten im Speicher
44   dc.l    io_Offset            ; Offsetwert für Device
48           io_SIZEOF
```

***io_Device, *io_Unit**
Beim Öffnen des Trackdisk-Device wird in io_Device ein Zeiger auf die Device-Struktur des TDD und in io_Unit ein Zeiger auf den Laufwerksport des angesprochenen Laufwerks (0-3) eingetragen. Zu diesen beiden Strukturen werden wir später in diesem Abschnitt kommen.

io_Command
Das Trackdisk-Device kennt die 8 Standard-Kommandos für Devices sowie einige devicespezifische Kommandos. Im Anschluß an die Struktur werden wir uns die Kommandos detailliert vornehmen, hier nur eine Kurzübersicht:

TDD-Kommando	Wert	Bedeutung
CMD_RESET	1	Device zurücksetzen
CMD_READ	2	Lese Daten von Diskette
CMD_WRITE	3	Schreibe Daten auf Diskette
CMD_UPDATE	4	Entleere Datenpuffer auf Disk
CMD_CLEAR	5	Lösche Datenpuffer ohne Rückschreiben
CMD_STOP	6	Device in Wartezustand bis 'Start'
CMD_START	7	Device reaktivieren nach 'Stop'
CMD_FLUSH	8	Alle IO-Kommandos abbrechen
TD_MOTOR	9	Laufwerksmotor ein- und ausschalten
TD_SEEK	10	Schreib/Lesekopf auf Spur positionieren
TD_FORMAT	11	Spuren formatieren
TD_REMOVE	12	Fehlerhafte Funktion (siehe weiter unten)!
TD_CHANGENUM	13	Anzahl Diskwechsel feststellen
TD_CHANGESTATE	14	Test, ob Disk im Laufwerk
TD_PROTSTATUS	15	Test, ob Disk schreibgeschützt
TD_RAWREAD	16	Lesen ohne anschließende Dekodierung

TD_RAWWRITE	17	Schreiben ohne vorhergehende Kodierung
TD_GETDRIVETYPE	18	Laufwerkstyp ermitteln
TD_GETNUMTRACKS	19	Maximalzahl Tracks ermitteln
TD_ADDCHANGEINT	20	Diskwechsel-Interrupt einbinden
TD_REMCHANGEINT	21	Diskwechsel-Interrupt entfernen

io_Flags
Die Flags spielen im Zusammenhang mit den Befehlen RAWREAD und RAWWRITE (siehe dort) und beim Öffnen des Devices eine Rolle.

TDD-Flag	Wert	Bedeutung
TD_ALLOW_NON_3_5	1	Zulassung von nicht-3 1/2 Zoll-Laufw.
IOTD_INDEXSYNC	16	Indexlock-Synchronisation einschalten
IOTD_WORDSYNC	32	$4489-Wortsynchronisation einschalten

Weitere Erklärungen folgen später.

io_Error
Wenn es bei einer Trackdisk-Operation zu einem Fehler kam, wird seine Nummer in diesem Eintrag zurückgegeben. Hier die Bedeutungen der Fehlernummern:

TDD-Fehler	Wert	Bedeutung
TDERR_NotSpecified	20	Allgemeiner Fehler beim Hardware-Zugriff
TDERR_NoSecHdr	21	Kein Sektorheader gefunden
TDERR_BadSecPreamble	22	Fehlerhafter Sektorvorspann
TDERR_BadSecID	23	Fehlerhafte Sektorkennmarke
TDERR_BadHdrSum	24	Sektorheader-Checksumme falsch
TDERR_BadSecSum	25	Checksumme über Blockdaten falsch
TDERR_TooFewSecs	26	Zu wenige Sektoren in einem Track
TDERR_BadSecHdr	27	Sektorheader fehlerhaft
TDERR_WriteProt	28	Schreibversuch auf schreibgeschützte Disk
TDERR_DiskChanged	29	Keine Disk im Laufwerk
TDERR_SeekError	30	Spur 0 kann nicht gefunden werden
TDERR_NoMem	31	Zu wenig Speicherplatz für Diskoperation
TDERR_BadUnitNum	32	Gewünschte Unitnummer nicht vorhanden
TDERR_BadDriveType	33	Laufwerkstyp wird vom TDD nicht unterstützt
TDERR_DriveInUse	34	Laufwerk wird schon benutzt
TDERR_PostReset	35	Device in Reset-Phase

Einige dieser Fehler, nämlich die Nummern 20 bis 27, haben nur dann Bedeutung, wenn Sie sich näher mit der Diskhardware-Programmierung beschäftigen wollen. Im später folgenden Abschnitt über die RAW-Befehle werden wir sie noch einmal kurz aufgreifen. Für brave TDD-User bedeuten sie alle schlicht: schwerer Fehler in der Datenstruktur der angesprochenen Spur. Ein solcher Fehler macht sich akustisch immer eindrucksvoll durch zwei langgezogene Kratzgeräusche des Schreib/Lese-Kopfes bemerkbar. Die Fehler ab Nummer 28 sind aber auch für uns von Interesse.

io_Actual
In diesem Langwort werden die Rückgabewerte von verschiedenen Kommandos abgelegt. Bei READ und WRITE z.B. steht hier die Anzahl der effektiv gelesenen bzw. geschriebenen Bytes; ebenso legen CHANGENUM, CHANGESTATE und PROTSTATUS ihre Rückmeldungen hier ab. In der anschließenden Besprechung der einzelnen Kommandos wird jeweils angegeben, wie der Actual-Eintrag belegt wird.

io_Length
Hier muß bei READ, WRITE und FORMAT die Anzahl der zu schreibenden Bytes abgelegt werden. Obwohl eine Angabe in Bytes verlangt wird, kann das TDD immer nur blockweise lesen und schreiben (Length ein Vielfaches von 512 sein), formatieren sogar nur trackweise (Length Vielfaches von 5632).

***io_Data**
Bei alle Kommandos, die einen Zeiger auf einen Speicherbereich erwarten (READ, WRITE, FORMAT etc.), muß dieser hier eingetragen werden.

io_Offset
Enthält die Blocknummer, auf die sich das Kommando beziehen soll (wird nicht immer benötigt). Auch hier muß die Angabe in Bytes erfolgen, das TDD kann jedoch nur an Blockgrenzen mit seiner Arbeit beginnen (READ und WRITE), für FORMAT und SEEK werden sogar Trackgrenzen erwartet.

10.3.3 Die Kommandos des Trackdisk-Device

Nun wollen wir die einzelnen Kommandos detailliert durchgehen. Wir stellen sie in einer ähnlichen Weise vor wie die Library-Routinen. Der Überschrifts-Kasten enthält den Namen, die Kommandonummer (die jeweils in den Struktureintrag **io_Command** muß!) und die Devicebezeichnung des Kommandos. Dann folgen, falls vorhanden, die Einträge und Offsets der IORequest-Struktur, die vor dem Aufruf mit Werten gefüllt werden müssen (erkennbar am '<'-Zeichen), dann, ebenso falls vorhanden, die Rückgabewerte ('>'-Zeichen) und schließlich eine Erklärung. Fangen wir an.

CMD_RESET	=	1	(Trackdisk-Device)

Erklärung Versetzt das Device in den 'Einschaltzustand', so, als wäre der Rechner gerade neu gestartet worden.

CMD_READ = 2 (Trackdisk-Device)

io_Length	36	<	Länge, der zu lesenden Daten in Byte. Muß ein Vielfaches von 512 sein, da das TDD nur ganze Blöcke bearbeiten kann.
*io_Data	40	<	Beginn des Lesepuffers im Speicher
io_Offset	44	<	Startblock für Lesevorgang. Achtung: Muß auch in Bytes angegeben werden, also als 'Startblock * 512'.
io_Error	31	>	Fehlernummer oder 0 für kein Fehler
io_Actual	32	>	Anzahl der wirklich gelesenen Bytes
Erklärung			Liest Daten von der Diskette in einen Puffer im Speicher. ACHTUNG: Bis einschließlich Kickstart 1.3 muß sich dieser Puffer im Chip-RAM befinden!

CMD_WRITE = 3 (Trackdisk-Device)

io_Length	36	<	Länge der zu schreibenden Daten in Byte. Muß ein Vielfaches von 512 sein, da das TDD nur ganze Blöcke bearbeiten kann.
*io_Data	40	<	Beginn des Datenpuffers im Speicher
io_Offset	44	<	Startblock für Schreibvorgang. Achtung: Muß auch in Bytes angegeben werden, also als 'Startblock * 512'.
io_Error	31	>	Fehlernummer oder 0 für kein Fehler
io_Actual	32	>	Anzahl der wirklich geschriebenen Bytes
Erklärung			Schreibt Daten aus dem Speicher auf die Diskette. ACHTUNG: Bis einschließlich Kickstart 1.3 muß sich der Datenpuffer im Chip-RAM befinden!

CMD_UPDATE = 4 (Trackdisk-Device)

io_Error	31	>	Fehlernummer oder 0
Erklärung			Schreibt den Inhalt des TDD-internen Datenpuffers auf die Disk.

Zum Begriff des Datenpuffers: Bei seinem Start reserviert das TDD einen Speicherbereich von festlegbarer Größe und benutzt ihn als Datenzwischenspeicher. Das bedeutet, daß, wenn z.B. ein Block gelesen werden soll, immer gleich die ganze Spur, in der sich der Block befindet, gelesen wird. Das ist aus hardware-technischen Gründen notwendig. Die Spur wird im Datenpuffer abgelegt. Soll nun ein weiterer Block aus der selben Spur gelesen werden (was ja meist recht wahr-

scheinlich ist), kann er sofort, ohne wirklichen Zugriff auf die Disk, aus dem Puffer geholt werden.

Auch beim Schreiben wird dieser Puffer verwendet. Schreibzugriffe werden zunächst nur in den Puffer ausgeführt. Erst, wenn der Pufferplatz für eine andere Spur benötigt oder das UPDATE-Kommando empfangen wird, wird der Pufferinhalt wirklich auf die Diskette gebracht. Sie sollten bei der Arbeit mit dem TDD also sicherstellen, daß nach dem letzten Schreibzugriff, dem keine sofortigen Lesezugriffe folgen, ein UPDATE ausgeführt wird, damit keine Daten im Puffer verschwinden und die Disk diese nie zu Gesicht bekommt.

CMD_CLEAR = 5 (Trackdisk-Device)

Erklärung Entleert den Inhalt des Datenpuffers, ohne ihn auf die Disk zu bringen.

Auch dieses Kommando bezieht sich auf den Datenpuffer. Es veranlaßt dessen Entleerung, ohne daß die Disk aktualisiert wird. Diesen Befehl können Sie benutzen, wenn Sie das physikalische Laden einer Spur von Disk erzwingen wollen, weil Sie nicht sicher sind, ob die Daten dieser Spur, die eventuell noch im Puffer stehen, aktuell sind.

CMD_STOP = 6 (Trackdisk-Device)

Erklärung Friert den Betrieb des TDD ein, bis das Kommando START empfangen wird.

Nach Ausführung dieses Kommandos führt das TDD keine weiteren Kommandos (bis auf START) mehr aus, speichert sie aber weiterhin in der Message-Warteschlange.

Bei Festplatten hat dieses Kommando noch eine weitere Funktion: Es veranlaßt das "Parken" der Platte, d.h. die Schreib/Lese-Köpfe werden in einen nicht zur Datenspeicherung verwendeten Bereich gefahren und der Motor der Platte ausgeschaltet.

CMD_START = 7 (Trackdisk-Device)

Erklärung Reaktiviert das TDD nach vorhergehender Desaktivierung durch STOP.

Nach dem STARTen werden alle in der Zwischenzeit eingegangenen IO-Requests, die sich in der Message-Schlange befinden, der Reihe nach abgearbeitet.

Auch hier gilt eine Besonderheit bei Festplatten: START hebt den Parkzustand auf, der Motor wird hochgefahren und die Platte ist wieder betriebsbereit.

CMD_FLUSH	**=**	**8**	**(Trackdisk-Device)**

Erklärung			Bricht den augenblicklichen IO-Prozeß ab und löscht auch alle weiteren, noch in der Message-Warteschlange befindlichen, IO-Requests.

TD_MOTOR	**=**	**9**	**(Trackdisk-Device)**

io_Length	**36**	<	Gewünschter Status des Laufwerksmotors: 0=aus, 1=ein
io_Actual	**32**	>	Motorstatus vor der Befehlsausführung: 0=aus, 1=ein
Erklärung			Schaltet den Motor des Laufwerks ein oder aus.

Dieses Kommando hat bei Festplatten keine Wirkung. Stattdessen müssen Sie die Kommandos STOP (parken) und START (entparken) verwenden.

TD_SEEK	**=**	**10**	**(Trackdisk-Device)**

io_Offset	**44**	<	Anzufahrende Zylindernummer (muß in Byte angegeben werden, also Zylinder * 11264)
Erklärung			Fährt den Schreib/Lese-Kopf zum angegebenen Zylinder.

Die Berechnung mittels Zylinder*11264 ergibt sich aus der Bytezahl, die ein Zylinder umfaßt: 11 Blöcke * 2 Seiten * 512 Bytes = 11264.

TD_FORMAT	**=**	**11**	**(Trackdisk-Device)**

io_Length	**36**	<	Anzahl der zu formatierenden Spuren in Byte (Berechnung: Spuren * 5632).
***io_Data**	**40**	<	Beginn der Daten im Speicher, die beim Formatieren auf die Disk geschrieben werden sollen
io_Offset	**44**	<	Startspur für Formatierung (Berechnung: Startspur * 5632).

io_Error 31 > Fehlernummer oder 0 für kein Fehler

Erklärung Formatiert Spuren einer Diskette und schreibt dabei Daten in sie. Bis Kickstart 1.3 müssen die Daten im Chip-RAM stehen.

Der Unterschied zwischen Formatieren und Schreiben ist folgender: Bei der Formatierung einer Spur wird ihre Datenstruktur komplett neu angelegt. Auf schon bestehende Daten kann dabei keine Rücksicht genommen werden, es muß die ganze Spur angelegt werden. Das ist der Grund, warum FORMAT nicht auf einzelne Blöcke anwendbar ist. Beim Schreiben aber wird die schon bestehende Spurstruktur berücksichtigt und die Daten werden gezielt in die gewünschten Blöcke geschrieben. Der Vorteil von FORMAT ist, daß er wesentlich schneller ausgeführt wird als WRITE.

Auf neue, bisher unbenutzte Disketten kann nicht mit WRITE zugegriffen werden, da noch keine Spurstruktur vorhanden ist. Diese muß zunächst mit FORMAT angelegt werden, erst ab dann ist WRITE möglich. Das gleiche gilt auch für Disketten, die "harte" Fehler aufweisen (TDD-Errors 20 bis 27) oder auf anderen Computersystemen formatiert wurden.

TD_REMOVE	**= 12**	**(Trackdisk-Device)**

***io_Data** 40 < Zeiger auf SoftInt-Struktur oder 0 zum Sperren des Interrupts

Erklärung Initialisiert oder sperrt einen Software-Interrupt, der beim Entfernen einer Diskette ausgelöst wird (für weitere Informationen über Interrupts siehe Exec-Kapitel)

TD_CHANGENUM	**= 13**	**(Trackdisk-Device)**

io_Actual 32 > Anzahl Diskwechsel seit Systemstart

Erklärung Ermittelt die Anzahl der seit Systemstart durchgeführten Diskwechsel (natürlich nur für die entsprechende TDD-Unit). Wichtig: Als Wechsel zählt sowohl das Einlegen als auch das Entfernen einer Disk!

TD_CHANGESTATE	**= 14**	**(Trackdisk-Device)**

io_Actual 32 > 0 - Disk im Laufwerk / <> 0 - Keine Disk

Erklärung Stellt fest, ob sich eine Disk im Laufwerk befindet. Achtung: Nach Einlegen einer neuen Disk kann es bis zu zwei Sekunden dauern, bis das TDD darauf reagiert!

TD_PROTSTATUS = 15 (Trackdisk-Device)

io_Actual 32 > 0 - Disk nicht schreibgeschützt / <> 0 - schreibgeschützt

Erklärung Stellt fest, ob die Disk im Laufwerk schreibgeschützt ist. Falls gar keine Disk im Laufwerk ist, hat der Rückgabewert natürlich keine Bedeutung.

Die Kommandos 16 und 17 werden nachfolgend in einem gesonderten Abschnitt besprochen.

TD_GETDRIVETYPE = 18 (Trackdisk-Device)

io_Actual 32 > Typ des angesprochenen Laufwerks

Erklärung Ermittelt den Typ des Laufwerks, auf den sich die IORequests beziehen (5 1/4 oder 3 1/2 Zoll-Laufwerk)

Laufwerkstyp	Wert	Bedeutung
DRIVE3_5	1	Normales 3 1/5 Zoll-Laufwerk
DRIVE5_25	2	5 1/4 Zoll-Laufwerk

TD_GETNUMTRACKS = 19 (Trackdisk-Device)

io_Actual 32 > Maximalzahl Spuren für das Laufwerk

Erklärung Stellt fest, wieviele Spuren das Laufwerk besitzt (beim Amiga 160: 80 Ober- und 80 Unterseiten).

TD_ADDCHANGEINT = 20 (Trackdisk-Device)

ACHTUNG: Dieses Kommando soll, ebenso wie TD_REMOVE, einen Interrupt installieren, der bei Diskwechseln aufgerufen wird. Aufgrund eines Fehlers im Betriebssystem wird aber anstatt des übergebenden SoftInt-Zeigers der Zeiger auf die

IORequest-Struktur in die Interruptliste aufgenommen, was beim nächsten Diskwechsel unweigerlich zu einem Absturz führt.

TD_REMCHANGEINT	**=**	**20**	**(Trackdisk-Device)**

Dieses Kommando soll einen mit TD_ADDCHANGEINT installierten Diskwechsel-Interrupt wieder entfernen. Da die ADDCHANGEINT-Routine jedoch fehlerhaft ist, wird die REMCHANGEINT-Routine wohl nie zum Einsatz kommen.

10.3.4 Die Kommandos RAWREAD und RAWWRITE

Nun zu den noch fehlenden Kommandos Nr. 16 und 17. Um sie zu verstehen, müssen wir einen kleinen Abstecher in die Floppyhardware machen.

Aufzeichnung und Kodierung

Die Aufzeichnung von Daten auf einer Diskette beruht auf der Magnetisierung ihrer Oberfläche, wobei zwei "gegenläufige" Magnetflußrichtungen eingesetzt werden. Ein 0-Bit wird als konstante Flußrichtung aufgezeichnet, ein 1-Bit als Flußrichtungswechsel. Beim Lesen erzeugt dann jeder Flußrichtungswechsel durch Induktion einen Stromimpuls im Schreib/Lese-Kopf. Diese Impulse werden als 1-Bits interpretiert, ein Ausbleiben des Impulses als 0-Bit.

So weit, so gut. Nun gibt es aber ein technisches Problem: Da 0-Bits als konstante Flußrichtung aufgezeichnet werden, müssen bei vielen aufeinanderfolgenden 0-Bits die einzelnen Bitzellen exakt gleich lang sein, damit die Floppy weiß, um wieviele 0-Bits es sich handelt, sprich der Floppymotor muß vollkommen konstant rotieren. Das aber ist technisch unmöglich, da jeder Motor Gleichlaufschwankungen unterworfen ist.

Ebenso dürfen nicht zu viele 1-Bits aufeinanderfolgen, da der Schreib/Lese-Kopf nicht in der Lage ist, zu viele schnelle Flußwechsel eindeutig zu erkennen.

Die aufzuzeichnenden Daten aber können jede beliebige Bitfolge enthalten. Die Daten müssen also vor der Aufzeichnung kodiert werden, und zwar so, daß die Anzahl aufeinanderfolgender 0- und 1-Bits nicht zu groß wird.

Der Amiga bedient sich dabei des sog. "MFM"-Kodierungsverfahrens (MFM = Modified Frequency Modulation, Verbesserte Frequenzmodulation). Hinter jedes Datenbit wird ein "Taktbit" in den Datenstrom eingefügt. Falls die beiden angrenzenden Bits die Kombinationen 0-0, 0-1 oder 1-0 aufweisen, wird ein 1-Taktbit eingefügt, bei der Kombination 1-1

ein 0-Taktbit. Somit ist gewährleistet, daß die Anzahl aufeinanderfolgender gleicher Bits im Rahmen bleibt.

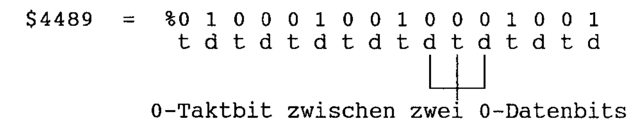
Auf diese Weise wird die Menge der aufzuzeichnenden Daten natürlich verdoppelt. Das MFM-Verfahren geht also nicht gerade sparsam mit dem Diskettenplatz um, dafür kann die Kodierung und Dekodierung aber recht schnell geschehen.

Blockheader und Blockdaten

Da der Schreib/Lese-Kopf nicht "weiß", wo auf der Diskette er sich gerade befindet, kann die Floppy primär Blöcke nicht gezielt einlesen. Das TDD liest bei "normalen" Lesekommandos immer einen ganzen Track ein und sucht sich dann den richtigen Bereich mit den gewünschten Blöcken heraus.

Damit nun in dem Datenwust die Blöcke eindeutig identifiziert werden können, findet sich vor den eigentlichen Daten jedes Blockes ein "Blockheader".

Der Header beginnt mit der "Preamble", einem kodierten 0-Wort (welches in MFM zu $aaaaaaaa wird). Dann folgt die sog. "Syncmarkierung", bestehend aus zwei $4489-Worten. Dieses Wort kann bei der MFM-Kodierung nicht vorkommen, da in seinem Bitmuster zwischen zwei 0-Datenbits ebenfalls ein 0-Taktbit stehen würde, was laut den MFM-Regeln verboten ist:



Dieses Wort ist somit gut als Startmarkierung für den Blockheader geeignet. Als nächstes folgt die Angabe, um welchen Block in welchem Track es sich handelt. Diese Werte sind für die Auswahl des richtigen Blocks beim Lesen wichtig. Dann folgen 16 unbenutzte Bytes (die in MFM zu 32 werden) und zum Abschluß zwei Checksummen, eine über den Blockheader und eine über den anschließend folgenden Datenbereich.

Die 16 unbenutzten Bytes werden gleich, im Zusammenhang mit den erweiterten TDD-Befehlen, eine Rolle spielen.

Nun verstehen Sie wahrscheinlich auch die TDD-Fehlermeldungen 20 bis 27. Sie beziehen sich auf die Einträge des Sektorheaders (Preamble, Sync, Checksummen).

Der Blockheader hat in der MFM-kodierten Form eine Länge von 64 Bytes. Die Daten selber belegen 1024 Bytes (512 Bytes hat ein Block normalerweise, durch MFM wird der Platz verdoppelt). Das macht insgesamt 11968 Bytes. Hinzu kommt noch eine 696 Bytes lange Lücke (unbenutzter Speicherbereich), womit wir eine Bytelänge von 12264 für jeden Track erreichen.

Mehr wollen wir zum Disk-Aufzeichnungsverfahren aber nicht sagen, Sie wissen jetzt alles, was zum Verständnis der Befehle RAWREAD und RAWWRITE nötig ist.

RAWREAD und RAWWRITE

TD_RAWREAD	**=**	**16**	**(Trackdisk-Device)**

io_Flags	**30**	**<**	Zugelassene Flags siehe unten
io_Length	**36**	**<**	Anzahl der aus dem Track zu lesenden Bytes (max. 12664)
***io_Data**	**40**	**<**	Start des Puffers im Speicher (muß immer im Chip-RAM liegen!)
io_Offset	**44**	**<**	Zu lesende Spur (Achtung: Angabe hier <u>nicht</u> in Bytes, sondern direkt als Spurnummer (0-159)!)
io_Error	**31**	**>**	Fehlernummer oder 0 für kein Fehler
Erklärung			Liest Daten von der Diskette, ohne sie anschließend zu dekodieren.

TD_RAWWRITE	**=**	**17**	**(Trackdisk-Device)**

io_Flags	**30**	**<**	Zugelassene Flags siehe unten
io_Length	**36**	**<**	Anzahl der in dem Track zu schreibenden Bytes (max. 12664)
***io_Data**	**40**	**<**	Start des Puffers im Speicher (muß immer im Chip-RAM liegen!)
io_Offset	**44**	**<**	Zu schreibende Spur (Achtung: Angabe hier <u>nicht</u> in Bytes, sondern direkt als Spurnummer (0-159)!)
io_Error	**31**	**>**	Fehlernummer oder 0 für kein Fehler
Erklärung			Schreibt Daten auf die Diskette, ohne sie vorher zu kodieren.

RAW-Flag	Wert	Bedeutung
IOTD_INDEXSYNC	16	Indexloch-Synchronisation einschalten
IOTD_WORDSYNC	32	$4489-Wortsynchronisation einschalten

Hierbei handelt es sich also um die "rohe" Form des Lese- bzw. Schreibbefehls. Er erwartet in io_Offset diesmal nicht die Blocknummer angegeben in Bytes, sondern direkt die Nummer des gewünschten Tracks (und damit der gewünschten Seite). Wie schon erwähnt, kann das Laufwerk nicht gezielt auf bestimmte Blöcke zugreifen, es kann nur da, wo der Lesekopf gerade steht, mit dem Lesen oder Schreiben beginnen. Der RAWREAD-Befehl kann auch nicht mehrere Tracks auf einmal

bearbeiten, die größte Byteanzahl ist die eines kompletten Tracks (12664).

Die Flags veranlassen das Laufwerk, mit dem Lesen zu warten, bis entweder das Indexloch (kleines Loch in der Diskettenscheibe, das als Orientierungspunkt dient) oder die $4489-Syncmarkierung des nächsten Blocks erreicht wird.

Die Daten liegen nach dem Lesen in MFM-kodierter Form vor. Das Heraussuchen des richtigen Blocks und das Dekodieren müssen Sie selbst übernehmen, ebenso das Kodieren vor dem Schreiben. Das ist der Grund, warum die RAW-Befehle des TDD gewöhnlich sehr selten eingesetzt werden. Wir wollen hier auch nicht näher auf sie eingehen.

Interessant ist vielleicht noch, daß es für die Floppyhardware keinen Unterschied zwischen Schreiben und Formatieren gibt. Sie formatiert quasi immer. Der Schreibbefehl des TDD liest nämlich zunächst den gewünschten Track, sucht den bzw. die richtigen Blöcke heraus, ersetzt ihre Inhalte durch die zu schreibenden Daten und schreibt dann den kompletten Track zurück.

10.3.5 Die erweiterten TDD-Kommandos

Hierbei handelt es sich um Sonderformen bestimmter normaler TDD-Kommandos. Um anzuzeigen, daß es sich um ein erweitertes Kommando handelt, muß die ursprüngliche Kommandonummer mit $8000 OR-verknüpft werden (32768 aufaddieren bzw. Bit 15 setzen). Folgende Kommandos existieren in der erweiterten Form:

Kommando	Nummer	Kommando	Nummer
ETD_READ	32770	ETD_WRITE	32771
ETD_UPDATE	32772	ETD_CLEAR	32773
ETD_MOTOR	32777	ETD_SEEK	32778
ETD_FORMAT	32779	ETD_RAWREAD	32784
ETD_RAWWRITE	32785		

Zur Benutzung dieser Befehle muß anstatt der IOStdReq-Struktur eine andere, um zwei Einträge größere verwendet werden: Die IOExtTD-Struktur.

Die IOExtTD-Struktur

```
00    ds.b    iotd_IOStdReq,48      ; IOStdReq-Struktur (wie gewöhnlich)
48    dc.l    iotd_Count            ; Zähler für Diskwechsel
52    dc.l   *iotd_SecLabel         ; 16-Byte-Datenpuffer (siehe unten)
56             iotd_SIZEOF
```

iotd_Count
Hier kann ein Zählerstand für den Diskwechsel-Zähler eingetragen werden. Das IO-Kommando wird dann nur ausgeführt,

wenn der wirkliche Diskwechsel-Zähler kleiner oder gleich dem angegebenen ist. Somit kann man z.B. sicherstellen, daß während einer Kette von Lese- und Schreibzugriffen nicht plötzlich die Diskette gewechselt wird und weitere Zugriffe auf eine andere Diskette ausgeführt werden.

iotd_SecLabel
Dieser Eintrag ermöglicht die Nutzung der 16 unbelegten, normalerweise nicht erreichbaren Bytes im Blockheader des angesprochenen Datenblocks. Wird hier ein Zeiger auf einen 16 Byte großen Speicherbereich eingetragen, so wird dieser Bereich im Falle eines Schreibzugriffes in den Blockheader geschrieben, beim Lesezugriff wird der Bereich mit dem Inhalt des Blockheaders gefüllt.

Die erweiterten Kommandos arbeiten also genauso wie ihre "nicht-erweiterten" Entsprechungen, nur daß die zwei Sonderfunktionen Diskwechselzähler-Kontrolle und Nutzung der 16 Bytes im Blockheader hinzukommen.

10.3.6 Der Aufbau des Trackdisk-Devices

Jedes Device hat bekanntlich eine Device-Struktur, die dem Aufbau der Library-Struktur entspricht. So auch das Trackdisk-Device. Allerdings ist dieser Aufbau, bis auf die Library-Struktur zu Beginn, nicht zwingend festgelegt, weshalb keine weiteren Angaben gemacht werden können.

Die Unit-Struktur

In den Eintrag *io_Unit der IORequest-Struktur schreibt das System beim Öffnen des Devices einen Zeiger auf die Unit-Struktur des angesprochenen Laufwerks. Von diesen Strukturen kann es bis zu 4 geben (logisch, es gibt ja maximal vier Laufwerke). Sie enthalten diverse, interessante Angaben für das Laufwerk, weshalb wir sie uns nun anschauen wollen.

Die Unit-Struktur (für Trackdisk-Device)

```
00    ds.b   tdu_MsgPort,34      ; ┐
34    dc.b   tdu_Flags           ; │  Allgemeine
35    dc.b   tdu_Pad             ; │ Unit-Struktur
36    dc.w   tdu_OpenCnt         ; ┘
38    dc.w   tdu_Comp01Track     ; Track für erste Precompensation
40    dc.w   tdu_Comp10Track     ; Track für zweite Precompensation
42    dc.w   tdu_Comp11Track     ; Track für dritte Precompensation
44    dc.l   tdu_StepDelay       ; Verzögerungszeit beim Steppen
48    dc.l   tdu_SettleDelay     ; Verzögerungszeit nach dem Steppen
52    dc.b   tdu_RetryCnt        ; Wiederholversuche bei Fehlern
53           tdu_SIZEOF
```

tdu_MsgPort
Den Beginn jeder Unit-Struktur macht der Message-Port, an den die IORequests für das betreffende Device geschickt werden.

tdu_Flags - tdu_OpenCnt
Werden nur zur Systemverwaltung benutzt und sind für uns uninteressant.

tdu_Comp01Track - tdu_Comp11Track
Da die einzelnen Spuren zur Diskettenmitte hin kürzer werden und damit langsamer unter dem Schreib/Lese-Kopf hindurchwandern, müssen die Daten auf weiter innen liegenden Spuren langsamer aufgezeichnet werden als auf außen liegenden. Die Floppyhardware kennt vier verschiedene Verzögerungszeitstufen: 0, 140, 280 und 560 Nanosekunden (=milliardstel Sekunden). Die CompTrack-Einträge geben an, ab welchem Track die jeweils nächste Verzögerungsstufe angewandt werden soll.

tdu_StepDelay
Hier steht ein Zahlenwert (standardmäßig 6000), der die Durchlaufzahl einer Verzögerungsschleife beim Versetzen des Schreib/Lese-Kopfes angibt. Dieses Versetzen geschieht, indem ein bestimmtes Hardwareregister aktiviert wird. Sobald die Floppy das Setzen erkennt, wird der Kopf um eine Spur verschoben. Dabei vergeht natürlich Zeit, weshalb das Kontrollprogramm eine Pause (Verzögerungsschleife) einlegen muß. Mit diesem Wert können Sie herumexperimentieren. Schreiben Sie z.B. einen kleineren Wert als 6000 hinein, geht das Steppen merklich schneller vonstatten. Es gibt aber einen gewissen Mindestwert, unter den Sie nicht gehen sollten (ca. 2000), da die Stepimpulse sonst zu schnell kommen und das Laufwerk nicht mehr "nachkommt" (Lesefehler).

tdu_SettleDelay
Hier steht ein Verzögerungswert für eine Leerschleife, die nach der Beendigung des Steppens eingelegt wird, damit sich das Floppysystem wieder einpendeln kann. Auch hier sind Experimente möglich.

tdu_RetryCnt
Hier wird angegeben, wie oft das Laufwerk bei einem Lesefehler einen erneuten Versuch starten soll (Standardwert 10), bevor ein Fehler ans TDD gemeldet wird, da auch auf eigentlich korrekten Disketten aufgrund von Motorschwankungen o.ä. schon mal Fehler auftreten können.

10.3.7 Demoprogramm

Ein Demoprogramm für das TDD darf natürlich nicht fehlen. Es erwartet in der Kommandozeile die Nummer des zu lesenden Laufwerks (0-3), liest den ersten Block von der Diskette und zeigt ihn auf dem Bildschirm an. Bei diesem ersten Block handelt es sich um den sog. "Boot-Block", dessen Bedeutung

im 12. Kapitel (DOS-Sonderteil, Abschnitt File System) noch genauer erklärt wird.

Wir werden hier nur den Teil, der für das Einladen zuständig ist, abdrucken. Die Bildschirm-Ausgaberoutine sparen wir uns.

```
* Programm 10.1 (Auszug): Demonstration für Trackdisk-Device

        ...

        move.l   ExecBase,a6      ; io.library öffnen
        moveq    #0,d0
        lea      IOName,a1
        jsr      OpenLib(a6)
        move.l   d0,IOBase
        beq      Error

        ...

main2:  moveq    #0,d1            ; Flags
        move.l   IOBase,a6        ; IO-Base
        lea      DevName,a0       ; DevName
        move.l   #48,d2           ; Größe
        jsr      CreateIOFast(a6)
        tst.l    d0
        beq      Error1           ; Fehler beim Erstellen
        move.l   d0,IOReq

        move.l   ExecBase,a6
        move.l   IOReq,a1
        move.w   #2,28(a1)        ; Kommando: Lesen
        move.l   #512,36(a1)      ; 512 Bytes
        move.l   #Data,40(a1)     ; Nach Adresse 'Data'
        move.l   #0,44(a1)        ; Offset = 0 (1. Block)
        jsr      DoIO(a6)         ; Ausführen
        tst.l    d0
        bne      Error2

        ...

Error2: move.l   ExecBase,a6
        move.l   IOReq,a1
        move.w   #9,28(a1)
        move.l   #0,36(a1)
        jsr      DoIO(a6)         ; Motor aus

        move.l   IOBase,a6
        move.l   IOReq,a0
        jsr      DeleteIOFast(a6)        ; Device schließen

Error1: move.l   ExecBase,a6
        move.l   IOBase,a1
```

```
        bset     #3,14(a1)      ; Expunge-Bit setzen
        jsr      CloseLib(a6)   ; IOLibrary schließen (entfernen)

Error:  moveq    #0,d0
        rts

* Datenbereich

DevName:        dc.b    "trackdisk.device",0
                even
IOName:         dc.b    "io.library",0
                even
DosName:        dc.b    "dos.library",0
                even

                ...

DosBase:        dc.l    0
IOBase:         dc.l    0
IOReq:          dc.l    0
```

Programm 10.1 (Auszug): Demonstration für Trackdisk-Device

10.4 Das Printer-Device

Nun wollen wir uns das Device ansehen, mit dem wir den Drucker ansteuern können. Im Vergleich zum Trackdisk-Device ist die ganze Sache hier aber viel einfacher.

Zunächst einmal ist wichtig, daß bei Benutzung des Printer-Devices die, den Drucker betreffenden Einstellungen im Preferences-Programm (Anschlußport, Druckertreiber, Druckmodus, Helligkeitswahl bei Grafikdruck etc.) von Bedeutung sind, da das Device auf sie zurückgreift.

Beim Öffnen des Devices sind die Angaben Unit und Flags ohne Bedeutung und sollten auf 0 stehen.

10.4.1 Ausdruck von normalem Text

Das Printer-Device kennt drei verschiedene Zugriffsarten und damit auch drei verschiedene Strukturen. Die erste Zugriffsart ist das normale Senden von auszudruckendem Text. Hierfür wird die IOStdRequest-Struktur verwendet:

Die IOStdRequest-Struktur (für Printer-Device)

```
00      ds.b     io_Message,20        ; Zugehörige Message-Struktur
20      dc.l    *io_Device            ; Zeiger auf Device
24      dc.l    *io_Unit              ; Zeiger auf Unit
28      dc.w     io_Command           ; Printer-Kommando
```

```
30    dc.b    io_Flags         ; nicht benutzt
31    dc.b    io_Error         ; Eventueller Fehler
32    dc.l    io_Actual        ; nicht benutzt
36    dc.l    io_Length        ; Anzahl der zu übertr. Zeichen
40    dc.l   *io_Data          ; Zeiger auf Daten im Speicher
44    dc.l    io_Offset        ; nicht benutzt
48            io_SIZEOF
```

io_Command
Von den Standard-Kommandos ist zur Druckausgabe natürlich nur WRITE sinnvoll. Das Printer-Device kennt aber noch 3 devicespezifische Kommandos, von denen zwei aber andere IO-Strukturen erwarten und daher etwas später erklärt werden.

Printer-Kommando	Wert	Bedeutung
CMD_WRITE	3	Normalen Text ausdrucken
PRD_RAWWRITE	9	Nicht-vorbehandelten Text ausdrucken
PRD_PRTCOMMAND	10	Standard-Druckerkommando senden
PRD_DUMPRPORT	11	Hardcopy eines Rastports ausdrucken

io_Error
Folgende Fehlermeldungen beim Drucken sind möglich:

Printer-Fehler	Wert	Bedeutung
PDERR_CANCEL	1	Drucker nicht druckbereit
PDERR_NOTGRAPHICS	2	Drucker kann keine Grafik drucken
PDERR_INVERTHAM	3	HAM-Grafik kann nicht invertiert werden
PDERR_BADDIMENSION	4	Ungültige Druck-Grafikgröße
PDERR_DIMENSIONOVFLOW	5	Druck-Grafikgröße zu hoch
PDERR_INTERNALMEMORY	6	Kein Speicher für interne Variablen
PDERR_BUFFERMEMORY	7	Kein Speicher für Druckpuffer

PDERR_CANCEL Wenn der Drucker eine gewisse Zeit lang (ca. 20 Sekunden) nicht auf ein gesandtes Kommando anspricht, weil z.B. das Papier ausgegangen ist, er auf Off-Line steht oder ganz einfach nicht eingeschaltet ist, wird ein System-Requester "Printer Trouble ... Retry/Cancel" angezeigt. Falls der User der Requester mit Retry beantwortet, wird der Zugriff erneut versucht, bei Cancel wird der IORequest mit dem Fehler PDERR_CANCEL zurückgesandt.

PDERR_NOTGRAPHICS Dieser Fehler tritt auf, wenn versucht wird, auf einem nicht-grafikfähigen Drucker (Typenrad o.ä.) das DumpRPort-Kommando (Rastport-Hardcopy) auszuführen.

PDERR_INVERTHAM Eine HAM-Grafik kann aufgrund ihres Aufbaus nicht invertiert ausgedruckt werden.

io_Length
Enthält die Länge des auszudruckenden Textes in Bytes. Falls hier eine -1 steht, muß der Text mit einem 0-Byte abgeschlossen sein.

***io_Data**
Zeiger auf den Textbeginn im Speicher.

CMD_WRITE	**=**	**3**	**(Printer-Device)**

io_Length	**36**	<	Bytelänge des auszudruckenden Texts (bei -1 muß der Text mit einem 0-Byte abgeschlossen sein).
***io_Data**	**40**	<	Startadresse des Textes im Speicher
io_Error	**31**	>	Fehlernummer oder 0, falls kein Fehler
Erklärung			Druckt vorbehandelten Text aus, d.h. bestimmte Steuerzeichen werden druckerspezifisch durch andere Zeichen ersetzt. Zum Ausdruck von reinem Text kann CMD_WRITE verwendet werden.

Die meisten Drucker kennen interne Steuersequenzen, die mit dem ESC-Zeichen (ASCII-Wert 27) beginnen. Ein Beispiel ist die Sequenz ESC-x1 ("x" und "1" als ASCII-Zeichen), die bei vielen Druckern den NLQ-Modus einschaltet. Solche sog. ESC-Sequenzen können mit dem Kommando CMD_WRITE nicht ausgegeben werden, da das ESC-Zeichen durch die Vorbehandlung durch andere Zeichen ersetzt wird. In diesem Fall muß das nächste Kommando, PRD_RAWWRITE, benutzt werden.

PRD_RAWWRITE	**=**	**9**	**(Printer-Device)**

io_Length	**36**	<	Bytelänge des auszudruckenden Texts (bei -1 muß der Text mit einem 0-Byte abgeschlossen sein).
***io_Data**	**40**	<	Startadresse des Textes im Speicher
io_Error	**31**	>	Fehlernummer oder 0, falls kein Fehler
Erklärung			Druckt nicht-vorbehandelten Text. Alle Zeichen werden genau so, wie sie kommen, zum Drucker geschickt.

Dieses Kommando kann auch zur Ausgabe von ESC-Sequenzen verwendet werden. Im Zweifelsfalle sollten Sie es anstelle von CMD_WRITE verwenden.

10.4.2 Ausgabe von Drucker-Steuerkommandos

Kommen wir zur zweiten Zugriffsart auf das Printer-Device und damit zur zweiten Struktur. Es gibt eine Reihe standardisierte Drucker-Steuerkommandos, die Funktionen wie Fett-

druck, Kursivdruck, NLQ etc. aktivieren. Sie sind druckerunabhängig und werden vom Device beim Senden an den jeweiligen, in den Preferences eingestellten, Drucker angepaßt.

Das Senden eines solchen Steuerkommandos geschieht mit dem Printer-Kommando PRD_PRTCOMMAND. Anstelle der IOStdReq-Struktur wird die folgende benötigt:

Die IOPrtCmdReq-Struktur (Printer Command Request)

```
00   ds.b   iopcr_IORequest,32     ; Eine IORequest-Struktur
32   dc.w   iopcr_PrtCommand       ; Steuerkommando
34   dc.b   iopcr_Parm0            ; Erster Parameter (falls nötig)
35   dc.b   iopcr_Parm1            ; Zweiter Parameter (falls nötig)
36   dc.b   iopcr_Parm2            ; Dritter Parameter (nicht benutzt)
37   dc.b   iopcr_Parm3            ; Vierter Parameter (nicht benutzt)
38          iopcr_SIZEOF
```

iopcr_IORequest
Vor der eigentlichen Struktur befindet sich eine IORequest-Struktur, also die einfachste Form der IO-Strukturen.

iopcr_PrtCommand
Folgende Steuerkommandos sind möglich. Falls ein kleines 'n' oder 'm' in einer Beschreibung auftritt, bedeutet dies, daß n bzw. m der Parameter des Kommandos ist, der in iopcr_Parm0 bzw. iopcr_Parm1 abgespeichert werden muß. In der "Wert"-Spalte stehen die Zahlen, die in iopcr_PrtCommand eingetragen werden müssen. In der Spalte "ESC-Sequenz" steht die Entsprechung des Steuerkommandos in ANSI-Form. Der Text ESC steht dabei für das ASCII-Zeichen 27, alle weiteren Zeichen müssen (bis auf ein kleines 'n' oder 'm', wenn in der Beschreibung n oder m als Parameter auftritt) exakt als ASCII-Zeichen übernommen werden.

Kommando	Wert	ESC-Sequenz	Bedeutung
aRIS	0	ESCc	Drucker-Reset ausführen
aRIN	1	ESC#1	Drucker initialisieren
aIND	2	ESCD	Zeilenvorschub
aNEL	3	ESCE	Wagenrücklauf + Zeilenvorschub
aRI	4	ESCM	Zeile nach oben
aSGR0	5	ESC[0m	Standard-Zeichensatz
aSGR3	6	ESC[3m	Kursivschrift ein
aSGR23	7	ESC[23m	Kursivschrift aus
aSGR4	8	ESC[4m	Unterstrichen ein
aSGR24	9	ESC[24m	Unterstrichen aus
aSGR1	10	ESC[1m	Fettdruck ein
aSGR22	11	ESC[11m	Fettdruck aus
aSFC	12	ESC[3nm	Vordergrundfarbe n (0-9) einstellen
aSBC	13	ESC[4nm	Hintergrundfarbe n (0-9) einstellen
aSHORP0	14	ESC[0w	Normale Druckbreite
aSHORP2	15	ESC[2w	Elite-Druckdichte ein
aSHORP1	16	ESC[1w	Elite-Druckdichte aus

aSHORP4	17	ESC[4w	Schmalschrift ein
aSHORP3	18	ESC[3w	Schmalschrift aus
aSHORP6	19	ESC[6w	Breitschrift ein
aSHORT5	20	ESC[5w	Breitschrift aus
aDEN6	21	ESC[6"z	Schattendruck ein
aDEN5	22	ESC[5"z	Schattendruck aus
aDEN4	23	ESC[4"z	Doppeldruck ein
aDEN3	24	ESC[3"z	Doppeldruck aus
aDEN2	25	ESC[2"z	NLQ ein
aDEN1	26	ESC[1"z	NLQ aus
aSUS2	27	ESC[2v	Hochstellen ein
aSUS1	28	ESC[1v	Hochstellen aus
aSUS4	29	ESC[4v	Tiefstellen ein
aSUS3	30	ESC[3v	Tiefstellen aus
aSUS0	31	ESC[0v	Normale Schriftstellung
aPLU	32	ESCL	Teillinie aufwärts
aPLD	33	ESCK	Teillinie abwärts
aFNT0	34	ESC(B	Zeichensatz USA
aFNT1	35	ESC(R	Zeichensatz Frankreich
aFNT2	36	ESC(K	Zeichensatz Deutschland
aFNT3	37	ESC(A	Zeichensatz England
aFNT4	38	ESC(E	Zeichensatz Dänemark 1
aFNT5	39	ESC(H	Zeichensatz Schweden
aFNT6	40	ESC(Y	Zeichensatz Italien
aFNT7	41	ESC(Z	Zeichensatz Spanien
aFNT8	42	ESC(J	Zeichensatz Japan
aFNT9	43	ESC(6	Zeichensatz Norwegen
aFNT10	44	ESC(C	Zeichensatz Dänemark 2
aPROP2	45	ESC[2p	Proportional ein
aPROP1	46	ESC[1p	Proportional aus
aPROP0	47	ESC[0p	Proportional-Einstellung löschen
aTSS	48	ESC[n E	Proportional-Offset n einstellen
aJFY5	49	ESC[5 F	Linksausrichtung
aJFY7	50	ESC[7 F	Rechtsausrichtung
aJFY6	51	ESC[6 F	Zentrierte Ausrichtung
aJFY0	52	ESC[0 F	Ausrichtung aus
aJFY2	53	ESC[2 F	Wortabstand (Zentrierung)
aJFY3	54	ESC[3 F	Buchstabenabstand (Ausrichtung)
aVERP0	55	ESC[0z	Zeilenabstand 1/8"
aVERP1	56	ESC[1z	Zeilenabstand 1/6"
aSLPP	57	ESC[nt	Seitenlänge n einstellen
aPERF	58	ESC[nq	Überspringe n (n>0) Zeilen
aPERF0	59	ESC[0q	Überspringen aus
aLMS	60	ESC#9	Linker Rand gesetzt
aRMS	61	ESC#0	Rechter Rand gesetzt
aTMS	62	ESC#8	Oberer Rand gesetzt
aBMS	63	ESC#2	Unterer Rand gesetzt
aSTBM	64	ESC[Pn;mr	Setze Ränder auf n oben, m unten
aSLRM	65	ESC[Pn;ms	Setze Ränder auf n links, m rechts
aCAM	66	ESC#3	Ränder löschen

aHTS	67	ESCH	Horizontal-Tabulator setzen
aVTS	68	ESCJ	Vertikal-Tabulator setzen
aTBC0	69	ESC[0g	Horizontal-Tabulator löschen
aTBC3	70	ESC[3g	Alle Horizontal-Tabulatoren löschen
aTBC1	71	ESC[1g	Vertikal-Tabulator löschen
aTBC4	72	ESC[4g	Alle Vertikal-Tabulatoren löschen
aTBCALL	73	ESC#4	Alle V- und H-Tabulatoren löschen
aTBSALL	74	ESC#5	Standard-Tabulatoren setzen
aEXTEND	75	ESC[Pn"x	Erweitertes Kommando n ausführen

iopcr_Parm0, iopcr_Parm1
Falls das Steuerkommando Parameter benötigt, müssen sie hier eingetragen werden.

iopcr_Parm2, iopcr_Parm3
Es sind Kommandos mit bis zu vier Parametern vorgesehen, derzeit gibt es aber nur welche mit maximal zweien. Diese beiden Einträge werden daher nicht benutzt.

PRD_PRTCOMMAND = 10 (Printer-Device)

PrtCommand	32	<	Auszuführendes Steuerkommando
Parm0	34	<	Erster Parameter (falls nötig)
Parm1	35	<	Zweiter Parameter (falls nötig)
Error	31	>	Fehlernummer oder 0, wenn kein Fehler
Erklärung			Sendet eins der eben beschriebenen Steuerkommandos zum Drucker.

10.4.3 Ausdrucken einer Rastport-Grafik

Die wahrscheinlich interessanteste Funktion, die das Printer-Device bietet, ist das Ausdrucken einer Bitmap-Grafik. Auf einem Schwarz/Weiß-Drucker werden dabei die verschiedenen Farben in mehr oder weniger dichte Punktmuster umgesetzt, die bei Betrachtung des Ausdrucks aus einiger Entfernung den Eindruck unterschiedlicher Helligkeiten erwecken. Auf einem Farbdrucker wird natürlich das beste Ergebnis erzielt, da die Farben fast originalgetreu wiedergegeben werden können.

Für die Funktion Rastport-Dump wird ebenfalls eine spezielle Struktur benötigt, die IODRPReq-Struktur:

Die IODRPReq-Struktur (Dump RastPort Request)

```
00    ds.b    iodrpr_IORequest     ; IORequest-Struktur
32    dc.l   *iodrpr_RastPort      ; Zeiger auf zu druckenden Rastport
36    dc.l   *iodrpr_ColorMap      ; Zeiger auf Farbtabelle des RPort
40    dc.l    iodrpr_Modes         ; View-Modus der Grafik
```

```
44      dc.w    iodrpr_SrcX          ; x-Start der Rastportgrafik
46      dc.w    iodrpr_SrcY          ; y-Start der Rastportgrafik
48      dc.w    iodrpr_SrcWidth      ; Breite der Rastportgrafik
50      dc.w    iodrpr_SrcHeight     ; Höhe der Rastportgrafik
52      dc.l    iodrpr_DestCols      ; Breite der Druckergrafik
56      dc.l    iodrpr_DestRows      ; Höhe der Druckergrafik
60      dc.w    iodrpr_Special       ; Optionsflags
62              iodrpr_SIZEOF
```

iodrpr_IORequest
Wie bei der IOPrtCmdReq-Struktur befindet sich vor der eigentlichen Struktur eine IORequest-Struktur, also die einfachste Form der IO-Strukturen.

***iodrpr_RastPort**
Ein Zeiger auf den Rastport, der die auszudruckende Grafik enthält.

***iodrpr_ColorMap**
Zeiger auf die Farbtabelle, welche die Farbeinstellungen für die Grafik enthält. Sie kann über den Zeiger **vp_ColorMap** im ViewPort der Grafik erreicht werden.

iodrpr_Modes
Hier muß der Viewmodus der Grafik (Hires, Interlace, HAM usw.) eingetragen werden. Dieser ist auch aus dem Viewport ersichtlich (Eintrag **vp_Modes**)

iodrpr_SrcX, iodrpr_SrcY, iodrpr_SrcWidth, iodrpr_SrcHeight
Die Startkoordinaten und Breite des Grafikteils, der ausgedruckt werden soll. Über diese Einträge ist es möglich, nur einen Teil des Rastports auszudrucken.

iodrpr_DestCols, iodrpr_DestRows
Die Breite und Höhe der Grafik beim Ausdruck (in Pixeln). Durch diese Werte ist es möglich, die Grafik auf dem Drucker verkleinert oder vergrößert auszugeben. Sogar eine getrennte Größenangabe für die horizontale und vertikale Richtung ist möglich. Falls der Ausdruck 1:1 erfolgen soll, können Sie diese beiden Werte auf 0 setzen und die Flag SPECIAL_FULLCOLS und SPECIAL_ASPECT benutzen.

iodrpr_Special
Hier können folgende Flags eingetragen werden:

RPortDump-Flag	Wert	Bedeutung
SPECIAL_MILCOLS	$001	DestCols ist in 1/1000" angegeben
SPECIAL_MILROWS	$002	DestRows ist in 1/1000" angegeben
SPECIAL_FULLCOLS	$004	Maximale Spaltenzahl beim Druck
SPECIAL_FULLROWS	$008	Maximale Zeilenzahl beim Druck
SPECIAL_FRACCOLS	$010	DestCols ist ein Bruchteil von FULLCOLS
SPECIAL_FRACROWS	$020	DestRows ist ein Bruchteil von FULLROWS
SPECIAL_CENTER	$040	Zentrierter Ausdruck
SPECIAL_ASPECT	$080	Verhältnis Breite-Höhe korrigieren
SPECIAL_DENSITY1	$100	Minimale Druckdichte

SPECIAL_DENSITY2	$200	Zweitniedrigste Druckdichte
SPECIAL_DENSITY3	$300	Zweithöchste Druckdichte
SPECIAL_DENSITY4	$400	Höchste Druckdichte

SPECIAL_MILCOLS	Ist dieses Flag gesetzt, wird die Angabe für DestCols als 1/1000" interpretiert, ansonsten als Pixelwert.
SPECIAL_MILROWS	Ist dieses Flag gesetzt, wird die Angabe für DestRows als 1/1000" interpretiert, ansonsten als Pixelwert.
SPECIAL_FULLCOLS	Die Grafik wird in maximaler Breite ausgedruckt. Die Angabe für DestCols wird ignoriert.
SPECIAL_FULLROWS	Die Grafik wird in maximaler Höhe ausgedruckt. Die Angabe für DestRows wird ignoriert.
SPECIAL_FRACCOLS	Der DestCols-Wert wird als Bruchteil der maximalen Druckbreite angenommen.
SPECIAL_FRACROWS	Der DestRows-Wert wird als Bruchteil der maximalen Druckhöhe angenommen.
SPECIAL_CENTER	Die Grafik wird zentriert ausgedruckt. Falls als Breite die maximale Druckbreite angegeben ist (oder das entsprechende Flag gesetzt ist), hat CENTER keine Funktion.
SPECIAL_ASPECT	Das Verhältnis Breite-Höhe wird automatisch gemäß der DestCols-Angabe korrigiert. Der Wert für DestRows wird ignoriert. In Kombination mit SPECIAL_FULLCOLS kann man eine Grafik mit maximaler Breite und angepaßtem Seitenverhältnis ausdrukken, ohne herumrechnen zu müssen.
SPECIAL_DENSITYn	Stellt die Druckdichte n (1-4) ein. Je höher die Dichte, desto besser die Auflösung, desto kleiner wird aber die Grafik.

PRD_DUMPRPORT = 11 (Printer-Device)

RastPort	**32**	<	Zeiger auf zu druckenden Rastport
ColorMap	**36**	<	Zeiger auf Farbtabelle des RPort
Modes	**40**	<	View-Modus der Grafik
SrcX	**44**	<	x-Start der Rastportgrafik
SrcY	**46**	<	y-Start der Rastportgrafik
SrcWidth	**48**	<	Breite der Rastportgrafik
SrcHeight	**50**	<	Höhe der Rastportgrafik
DestCols	**52**	<	Breite der Druckergrafik
DestRows	**56**	<	Höhe der Druckergrafik
Special	**60**	<	Optionsflags
Error	**31**	>	Fehlernummer oder 0, wenn kein Fehler

Erklärung Druckt die angegebene Rastport-Grafik mit den angegebenen Parametern aus (Hardcopy).

10.4.4 Demoprogramm: Hardcopy

Als Beispielprogramm nehmen wir uns das interessanteste Feature des Printer-Device vor: die DumpRPort-Funktion. Wir erweitern das Anzeigeprogramm für IFF-Grafiken aus dem Graphics-Kapitel um eine Routine, welche die eingeladene und dargestellte Grafik per Hardcopy ausdruckt. Der Name der Grafik wird wieder in der Kommandozeile erwartet.

```
* Programm 10.2 (Auszug): Ausdrucken einer IFF-Grafik

        ...                             ; IFF-Grafik ist als Screen darge-
                                        ; stellt.

main4:  bsr       printscreen           ; Zur Ausdruck-Routine

        ...

printscreen:
        move.l    ExecBase,a6           ; IO-Library öffnen
        lea       ioname,a1
        clr.l     d0
        jsr       OpenLib(a6)
        tst.l     d0                    ; Fehler?
        beq       ps2

        move.l    d0,iobase
        move.l    d0,a6

        lea       prtname,a0            ; Name des Devices
        clr.l     d0                    ; Keine Unit
        clr.l     d1                    ; Keine Flags
        moveq     #62,d2                ; Strukturlänge 62
        jsr       CreateIOFast(a6)            ; Dev. öffnen
        tst.l     d0                    ; Fehler?
        beq       ps1

        move.l    d0,prtreq
        move.l    d0,a1                 ; Printer-Request

        move.w    #11,28(a1)            ; Kommando: DumpRPort
        move.l    scr,32(a1)            ; Zeiger auf Screen
        add.l     #$54,32(a1)           ; Screen+$54 = RastPort
        move.l    scr,a0                ; Screen-Zeiger
        add.l     #$2c,a0               ; Screen+$$2c = ViewPort
        move.l    4(a0),36(a1)          ; Zeiger auf Colormap
        clr.l     d0
        move.w    sview,d0              ; Viewmodus
        move.l    d0,40(a1)
        move.w    #0,44(a1)             ; Start-x
        move.w    #0,46(a1)             ; Start-y
        move.w    swidth,48(a1)         ; Breite Bildschirmgrafik
        move.w    sheight,50(a1)        ; Höhe Bildschirmgrafik
```

```
        clr.l     52(a1)          ; Breite u. Höhe Druckergrafik
        clr.l     56(a1)          ; nicht nötig
        move.w    #$84,60(a1)     ; Flags: FullCols, Aspect

        move.l    ExecBase,a6
        jsr       DoIO(a6)        ; Druck ausführen

        tst.b     31(a1)          ; Fehler?
        beq       ps3

        move.l    dosbase,a6      ; Meldung "Device-Fehler"
        lea       txt4,a0
        bsr       print

ps3:    move.l    iobase,a6       ; Dev. schließen
        move.l    prtreq,a0
        jsr       DeleteIOFast(a6)

ps1:    move.l    a6,a1           ; IO-Lib schließen
        move.l    ExecBase,a6
        jsr       CloseLib(a6)

ps2:    rts

        ...

* Datenbereich

dosname:        dc.b    "dos.library",0
                even
intname:        dc.b    "intuition.library",0
                even
gfxname:        dc.b    "graphics.library",0
                even
ioname:         dc.b    "io.library",0
                even
dosbase:        dc.l    0
intbase:        dc.l    0
gfxbase:        dc.l    0
iobase:         dc.l    0

prtname:        dc.b    "printer.device",0
                even
prtreq:         dc.l    0

                ...

txt4:           dc.b    "Printer-Device meldete Fehler!",10,0
```

Programm 10.2 (Auszug): Ausdrucken einer IFF-Grafik

10.5 Das Console-Device

Das Console-Device übernimmt die Ein- und Ausgabefunktionen. Dabei ist die Arbeit des Console-Device immer an ein Fenster gebunden, über welches kommuniziert werden soll. Für dieses Fenster wurde beim Öffnen des Devices eine spezielle Console-Unit-Struktur angelegt, die nur für dieses Fenster zuständig ist. Die Struktur enthält, neben den Daten für das Fenster, auch noch einen Puffer für die Daten (liegt im privaten Bereich), die gelesen worden sind. Diese Daten, die schon mit Hilfe der "keyboard.resource" in ASCII-Zeichen umgewandelt worden sind, können mit Hilfe der cmd_Read-Anweisung ausgelesen werden.

ConUnit-Struktur:

```
000   dc.l   *mp_Succ            ;┐
004   dc.l   *mp_Pred            ;│
008   dc.b    mp_Type            ;│ MessagePort
009   dc.b    mp_Pri             ;│
010   dc.l   *mp_Name            ;│
014   dc.b    mp_Flags           ;│
015   dc.b    mp_SigBit          ;│
016   dc.l    mp_SigTask         ;│
020   dc.l   *lh_Head            ;│
024   dc.l   *lh_Tail            ;│
028   dc.l   *lh_TailPred        ;│
032   dc.b    lh_Type            ;│
033   dc.b    lh_Pad             ;┘

034   dc.l   *cu_Window          ; Zeiger auf Fenster
038   dc.w    cu_XCP             ;┬─ Zeichenposition
040   dc.w    cu_YCP             ;┘
042   dc.w    cu_XMax            ;┬─ Max. Zeichenposition
044   dc.w    cu_YMax            ;┘
046   dc.w    cu_XRSize          ;┬─ Größe des Zeichenrasters
048   dc.w    cu_YRSize          ;┘
050   dc.w    cu_XROrigin        ;┬─ Anfangspunkt
052   dc.w    cu_YROrigin        ;┘
054   dc.w    cu_XRExtant        ;┬─ Ausdehnung
056   dc.w    cu_YRExtant        ;┘
058   dc.w    cu_XMinShrink      ;┬─ Min. Fensterausdehnung
060   dc.w    cu_YMinShrink      ;┘
062   dc.w    cu_XCCP            ;┬─ Position des Cursors
064   dc.w    cu_YCCP            ;┘

066   dc.l   *km_LoKeyMapTypes   ;┐
070   dc.l   *km_LoKeyMap        ;│
074   dc.l   *km_LoCapsable      ;│
078   dc.l   *km_LoRepeatable    ;│    KeyMap-Struktur
082   dc.l   *km_HiKeyMapTypes   ;│      (aktuelle)
086   dc.l   *km_HiKeyMap        ;│
090   dc.l   *km_HiCapsable      ;│
094   dc.l   *km_HiRepeatable    ;┘
```

```
098     ds.w    cu_TabStops,80      ; Tabulatoren
258     dc.b    cu_Mask             ;┐
259     dc.b    cu_FgPen            ;│
260     dc.b    cu_BgPen            ;│
261     dc.b    cu_AOLPen           ;│
262     dc.b    cu_DrawMode         ;│
263     dc.b    cu_AreaPtSz         ;│
264     dc.l    cu_AreaPtrn         ;│
268     ds.b    cu_Minterms,8       ;├ Rastport-Attribute
276     dc.l   *cu_Font             ;│
280     dc.b    cu_AlgoStyle        ;│
281     dc.b    cu_TxFlags          ;│
282     dc.w    cu_TxHeight         ;│
284     dc.w    cu_TxWidth          ;│
286     dc.w    cu_TxBaseline       ;│
288     dc.w    cu_TxSpacing        ;┘

290     ds.b    cu_Modes,3          ;┐ Modes und
293     ds.b    cu_RawEvents,3      ;┘ RawEvents
296             cu_SIZEOF
```

Die, in der ConUnit-Struktur abgelegten Werte beziehen sich, wie schon erwähnt, auf ein bestimmtes Fenster. Dieses Fenster, bzw. die Adresse auf dessen Window-Struktur, muß in die IORequest-Struktur eingetragen werden. Nachdem die Struktur eingerichtet worden ist (OpenDevice), enthält der Eintrag io_Unit die Adresse der neu angelegten ConUnit-Struktur und der Eintrag io_Device die Adresse der Library-Struktur des Devices.

Neben den Standard-Device-Funktionen gibt es noch vier Spezialfunktionen, mit denen man die Tastaturtabelle, die beim Umwandeln der RAW-Codes in ASCII-Codes benutzt werden soll, setzen bzw. erfragen kann.

cd_AskKeyMap = 9 (Console-Device)

io_Length	**36**	<	Länge der KeyMap-Struktur.
io_Data	**40**	<	Zeiger auf Datenbereich für KeyMap-Struktur.
io_Error	**31**	>	Fehlernummer.
Erklärung			Es wird die momentan gesetzte Tastaturtabelle (KeyMap-Struktur) in den angegebenen Bereich kopiert.

cd_SetKeyMap = 10 (Console-Device)

io_Length	**36**	<	Länge der KeyMap-Struktur.
io_Data	**40**	<	Zeiger auf Datenbereich für KeyMap-Struktur, die gesetzt werden soll.

io_Error **31** > Fehlernummer.

Erklärung Es wird die angegebenen KeyMap-Struktur gesetzt.

cd_AskDefaultKeyMap	=	**11**	**(Console-Device)**

io_Length **36** < Länge der KeyMap-Struktur.
io_Data **40** < Zeiger auf Datenbereich für KeyMap-Struktur.

io_Error **31** > Fehlernummer.

Erklärung Es wird die Default-Tastaturtabelle in den angegebenen Bereich kopiert.

cd_SetDefaultKeyMap	=	**12**	**(Console-Device)**

io_Length **36** < Länge der KeyMap-Struktur.
io_Data **40** < Zeiger auf Datenbereich für KeyMap-Struktur.

io_Error **31** > Fehlernummer.

Erklärung Es wird die Default-Tastaturtabelle in den angegebenen Bereich kopiert.

10.5.1 Die Keymap-Struktur

Die oben angesprochene KeyMap-Struktur, anhand derer die RAW-Codes in ASCII-Zeichen konvertiert werden, besteht aus acht Langworten. Dabei sind die ersten vier Zeiger für die RAW-Codes $00 bis $3f (Lo) und die letzten vier für die RAW-Codes $40 bis $67 (Hi) verantwortlich.

KeyMap-Struktur:

```
00      dc.l    *km_LoKeyMapTypes      ;┐ Zeiger auf Typentab.
04      dc.l    *km_LoKeyMap           ;│ Lo Zeiger auf KeyMap
08      dc.l    *km_LoCapsable         ;│ $00-$3f Capsable-Werte
12      dc.l    *km_LoRepeatable       ;┘ Repeatable-Werte

16      dc.l    *km_HiKeyMapTypes      ;┐ Zeiger auf Typentab.
20      dc.l    *km_HiKeyMap           ;│ Hi Zeiger auf KeyMap
24      dc.l    *km_HiCapsable         ;│ $40-$67 Capsable-Werte
28      dc.l    *km_HiRepeatable       ;┘ Repeatable-Werte
32               km_SIZEOF
```

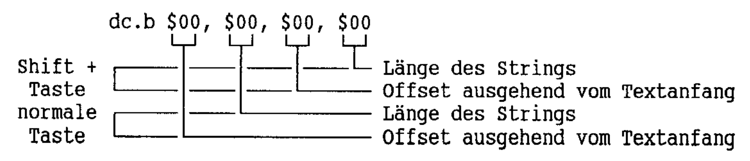
***km_LoKeyMapTypes/*km_HiKeyMapTypes**
Der erste Langwort-Wert (km_Lo-/km_HIKeyMapTypes) enthält die Adresse einer Tabelle, die für jeden RAW-Code (0-$3f oder $40-$67) einen Byte-Wert enthält, in dem der Typ der Taste abgelegt worden ist.

Name	Wert	Bedeutung
KC_NORMAL	$00	Taste hat nur eine Belegung
KC_SHIFT	$01	Shift-Taste
KC_ALT	$02	Alt-Taste
KC_CONTROL	$04	Ctrl-Taste
KC_VANILLA	$07	Ctrl-Taste wird normal behandelt
KC_DOWNUP	$08	Taste wurde losgelassen
KC_DEAD	$20	keine Reaktion
KC_STRING	$40	Taste gibt eine Zeichenkette aus

***km_LoKeyMap/*km_HiKeyMap**
Der nächste Zeiger (*km_Lo-/*km_HiKeyMap) verweist auf eine Tabelle, in der für jeden RAW-Code vier Bytes abgelegt sind. Diese vier Bytes werden je nach Kombination der KC-Flags (*km_LoKeyMapTypes/*km_HiKeyMapTypes) benutzt. Dabei soll die nächste Tabelle die Belegung der vier Bytes, bei den erlaubten Typ-Kombinationen, zeigen.

Kombination	Wert	1.Byte	2.Byte	3.Byte	4.Byte
KC_DEAD	$20	-	-	-	-
KC_NOQUAL	$00	-	-	-	normal
KC_SHIFT	$01	-	-	S	normal
KC_ALT	$02	-	-	A	normal
KC_CONTROL	$04	-	-	C	normal
KC_ALT+KC_SHIFT	$03	S + A	A	S	normal
KC_CONTROL+KC_ALT	$06	C + A	C	A	normal
KC_CONTROL+KC_SHIFT	$05	C + S	C	S	normal
KC_VANILLA	$07	S + A	A	S	normal
KC_STRING	$40	Zeiger auf Zeichenkette			

Bei der KC_STRING-Kombination wird, wie in der Tabelle zu sehen ist, der Zeiger auf die String-Daten erwartet, die ausgegeben werden sollen. Dabei handelt es sich nicht nur um eine Zeichenkette, die ausgegeben werden soll. Vielmehr wird auf ein Langwort-Wert gezeigt, welcher den Offset und die Länge des Textes, der ausgegeben werde soll, enthält.




```
String01:   dc.b        $02,$04,$02,$06,$9B,$41,$9B,$54
```

Wie man an dem Beispiel sieht, wird bei "normaler" Taste die CSI-Sequenz $9b, $41 und bei Shift $9B, $54 ausgegeben.

Die letzten beiden Addressen gehören zu zwei Tabellen, die sich aus acht Bytes zusammensetzen. Jedes Bit dieser acht Bytes repräsentiert eine Taste, also insgesamt 64 Tasten.

***km_LoCapsable/*km_HiCapsable**
Bei der *km_Lo-/*km_HiCapsable-Tabelle entscheiden die Bits, ob die jeweilige Taste durch den CapsLock-Modus tangiert wird oder nicht. Durch diese Einstellung kann man z.B. verhindern, daß während des CapsLock-Modus auch die Zahlen-Tasten "geshiftet" werden.

***km_LoRepeatable/*km_HiRepeatable**
Mit den Werten der *km_Lo-/*km_HiRepeatable-Tabelle kann bestimmt werden, ob bei längerem Drücken der Taste die Ausgabe wiederholt werden soll.

Abschließend möchte ich noch erwähnen, daß die Tastaturtabellen-Dateien, die sich im Verzeichnis "Devs:KeyMaps/" befinden, grundsätzlich den gleichen Aufbau haben. Bei ihnen ist lediglich eine Node-Struktur vor die KeyMap-Struktur gelegt worden. Wie bei vielen anderen Dateien wird auch die KeyMap-Datei als ausführbares Programm gespeichert, damit die Zeiger auf die Tabellen richtig relokiert werden. So kann man z.B. eine externe KeyMap-Datei mit LoadSeg einladen und im eigenem Programm verwenden.

Nun aber zurück zum Console-Device. Will man also über ein Fenster die Ein- und Ausgabe ablaufen lassen, so muß man eine Write- und eine ReadIORequest-Struktur installieren. Damit aber nicht zwei ConUnit-Strukturen angelegt werden, öffnet man nur einmal das Device und überträgt die relevanten Werte in die zweite Struktur. Danach kann man beide IO-Request-Strukturen wie gewöhnlich benutzen.

Aus Platzgründen haben wir nur einen Ausschnitt aus dem Demonstrationsprogramm abgedruckt. Es soll lediglich gezeigt werden, welche Schritte notwendig sind, um Ein- und Ausgabeoperationen über das Console-Device abzuwickeln.

```
*
* Kapitel 12
* Demonstrationsprogramm für das "console.device"
*

Start:
        ...

        move.l   IOBase,a6
```

```
moveq    #0,d0
sub.l    a0,a0
jsr      CreatePort(a6) ; Reply-Port für Read erstellen

move.l   d0,a0
move.l   #48,d0
jsr      CreateIOReq(a6)        ; IOReq für Read erstellen
move.l   d0,IORReq

move.l   IOBase,a6
moveq    #0,d0
sub.l    a0,a0
jsr      CreatePort(a6) ; Reply-Port für Write erstellen

move.l   d0,a0
move.l   #48,d0
jsr      CreateIOReq(a6)        ; IOReq für Write erstellen
move.l   d0,IOWReq

move.l   ExecBase,a6
move.l   IORReq,a1
move.l   WindowHD,40(a1)        ; Zeiger auf WindowHD eintragen
moveq    #0,d0          ; Unit
moveq    #0,d1          ; Flags
lea      ConName,a0
jsr      OpenDev(a6)    ; Device öffnen

move.l   IORReq,a0      ; Zeiger auf IO-Read-Request
move.l   IOWReq,a1      ; Zeiger auf IO-Write-Request
move.l   20(a0),20(a1)  ; Device und
move.l   24(a0),24(a1)  ; Unit übertragen

...

move.l   ExecBase,a6
move.l   IORReq,a1
jsr      CloseDev(a6)   ; Device schließen

move.l   IOBase,a6
move.l   IOWReq,a0
move.l   14(a0),a0
jsr      DeletePort(a6) ; Write Reply-Port löschen

move.l   IOWReq,a0
jsr      DeleteIOReq(a6)        ; Write IOReq löschen

move.l   IORReq,a0
move.l   14(a0),a0
jsr      DeletePort(a6) ; Read Reply-Port löschen

move.l   IORReq,a0
jsr      DeleteIOReq(a6)        ; Read IOReq löschen
...
```

Programm 10.3: Benutzung des Console-Device

10.5.2 Die "conio.library"

Wie man an dem Demonstrationsprogramm sieht, ist die Benutzung des Console-Devices sehr umständlich und platzintensiv. Deshalb haben wir uns entschieden, einige nützliche Funktionen in einer weiteren Library ("conio.library") zusammenzufassen. Diese Funktionen sollen zunächst als Demonstration des Umgangs mit dem Device verstanden werden. Aber auch die Realisierung von Ein-/Ausgabe bezogenen Programmen soll dadurch ermöglicht werden. Hier wollen wir uns nicht die einzelnen Routinen ansehen, da sie auf der Programmdiskette ausreichend kommentiert sind. Lediglich der Aufruf der Funktionen soll erläutert werden.

InstallInp	**=**	**-30 (ConIO-Library)**

***WinArgs** a0 < Zeiger auf eine Window-Struktur.

***ConIOUnit** d0 > Zeiger auf die angelegte ConIO-Unit-Struktur (oder Null bei einem Fehler).

Erklärung Durch die Funktion InstallInp wird das, durch die übergebene Struktur definierte Fenster geöffnet und eine Read-IORequest-Struktur sowie eine Write-IORequest-Struktur erstellt. Die Zeiger auf die einzelnen Strukturen sind in einer ConIO-Unit-Struktur abgelegt, auf die in d0 ein Zeiger zurückgegeben wird.

ConIO-Unit-Struktur:

```
00    dc.l   *ciou_Window        ; Zeiger auf Window-Struktur
04    dc.l   *ciou_RReq          ; Zeiger auf RIORequest-Struktur
08    dc.l   *ciou_WReq          ; Zeiger auf WIORequest-Struktur
12            ciou_SIZEOF
```

RemoveInp	**=**	**-36 (ConIO-Library)**

***ConIOUnit** a0 < Zeiger auf eine ConIOUnit-Struktur, die entfernt werden soll.

Erklärung Mit RemoveInp kann man die angegebene ConIOUnit-Struktur und alle damit verbundenen Strukturen (MsgPort-Strukturen, Window-Strukturen, IOReq-Strukturen) wieder freigeben. Dabei wird das Fenster automatisch geschlossen.

ReadChar	=	-42 (ConIO-Library)

*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Char d0 > Gelesenes Zeichen.

Erklärung Es wird solange gewartet, bis ein Zeichen von der Tastatur eingelesen worden ist.

WriteChar	=	-48 (ConIO-Library)

*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
Char d0 < Byte-Wert, der ausgegeben werden soll.

Erklärung Das in d0 gespeicherte Zeichen wird ausgegeben.

ReadCrL	=	-54 (ConIO-Library)

*Buffer a0 < Zeiger auf einen Pufferbereich für die zu lesenden Zeichen.
*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Count d0 > Anzahl der gelesenen Zeichen.

Erklärung Durch die Funktion ReadCrL wird eine Zeichenkette eingelesen. Dabei schließt die Return-Taste die Eingabe ab. Danach wird der Cursor in die erste Spalte der gleichen Zeile gesetzt.

ReadStr	=	-60 (ConIO-Library)

*Buffer a0 < Zeiger auf einen Pufferbereich für die zu lesenden Zeichen.
*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Count d0 > Anzahl der gelesenen Zeichen.

Erklärung Durch die Funktion ReadStr wird eine Zeichenkette eingelesen. Dabei schließt die Return-Taste die Eingabe ab. Danach wird der Cursor in die erste Spalte der nächsten Zeile gesetzt.

WriteLeL	= -66 (ConIO-Library)

*String a0 < Zeiger auf eine mit einem Null-Byte abgeschlossene Zeichenkette.
*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
Count d0 < Anzahl der Zeichen, die ausgegeben werden sollen.

Erklärung Die angegebene Zeichenkette wird ausgegeben. Dabei muß ihre Länge im Datenregister 0 übergeben werden.

WriteStr	= -72 (ConIO-Library)

*String a0 < Zeiger auf eine mit einem Null-Byte abgeschlossene Zeichenkette.
*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Erklärung Die angegebene Zeichenkette wird ausgegeben. Dabei muß ihr Ende durch ein Null-Byte bestimmt sein.

WriteXY	= -78 (ConIO-Library)

*String a0 < Zeiger auf eine mit einem Null-Byte abgeschlossene Zeichenkette.
*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
XPos d0 < X-Position
YPos d1 < Y-Position

Erklärung Zunächst wird der Cursor an die Stelle d0/d1 gesetzt, dann wird die definierte Zeichenkette ausgegeben (WriteStr).

WriteDec	= -84 (ConIO-Library)

*ConIOUnit a5 < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
Value d0 < Hex-Wert (Wort)

Erklärung Die in d0 angegebene Zahl wird in einen Dez-String konvertiert und ausgegeben.

WriteHex	=	**-90 (ConIO-Library)**

*ConIOUnit	a5	<	Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
Value	d0	<	Hex-Wert (Langwort)
Erklärung			Die in d0 angegebene Zahl wird in einen Hex-String konvertiert und ausgegeben.

GotoXY	=	**-96 (ConIO-Library)**

*ConIOUnit	a5	<	Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
XPos	d0	<	X-Position
YPos	d1	<	Y-Position
Erklärung			Der Cursor wird an die Stelle d0/d1 gesetzt.

ScrollUp	=	**-102 (ConIO-Library)**

*ConIOUnit	a5	<	Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
NumLine	d0	<	Anzahl der Zeilen, um die der Bildschirminhalt hochgerollt werden soll.
Erklärung			Der Bildschirminhalt wird um "NumLine" Zeilen hochgerollt.

ScrollDown	=	**-108 (ConIO-Library)**

*ConIOUnit	a5	<	Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
NumLine	d0	<	Anzahl der Zeilen, um die der Bildschirminhalt runtergerollt werden soll.
Erklärung			Der Bildschirminhalt wird um "NumLine" Zeilen runtergerollt.

DelLine	=	**-114 (ConIO-Library)**

*ConIOUnit	a5	<	Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
NumLine	d0	<	Anzahl, der zu löschenden Zeilen.
Erklärung			Löscht ab der momentanen Position "NumLine" Zeilen.

InsLine	=	**-120 (ConIO-Library)**

***ConIOUnit** **a5** < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.
NumLine **d0** < Anzahl einzufügender Zeilen.

Erklärung Fügt "NumLine" Zeilen an der momentanen Position ein.

DelEOD	=	**-126 (ConIO-Library)**

***ConIOUnit** **a5** < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Erklärung Löscht ab der momentanen Position bis zum Fensterende alle Zeichen (Delete end of display).

DelEOL	=	**-132 (ConIO-Library)**

***ConIOUnit** **a5** < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Erklärung Löscht ab der momentanen Position bis zum Zeilenende alle Zeichen (Delete end of Line).

CursorOn	=	**-138 (ConIO-Library)**

***ConIOUnit** **a5** < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Erklärung Schaltet den Cursor an.

CursorOff	=	**-144 (ConIO-Library)**

***ConIOUnit** **a5** < Zeiger auf die ConIO-Unit-Struktur, die zur Ausgabe benutzt werden soll.

Erklärung Schaltet den Cursor ab.

Sehr auffällig an den Funktionsaufrufen ist, daß immer der Zeiger auf die ConIO-Unit benötigt wird. Wir haben dabei extra darauf geachtet, daß immer das Adreßregister a5 benutzt wird. So kann der Zeiger immer dort belassen werden und muß nicht bei jedem Funktionsaufruf erneut eingesetzt werden.

Zur Demonstration der "conio.library" finden sie auf der beiliegenden Diskette ein gut dokumentiertes Programm (PRG_10_4.S). Es handelt sich dabei um ein Trainingsprogramm für blindes Zehnfinger-Schreiben. Dennoch soll sich nun noch ein kleines Demonstrationsprogramm anschließen, welches lediglich den einfachen Umgang mit der conio.library dokumentieren soll.

```
*
* Kapitel 10
* Demonstrationsprogramm für die conio.library
*

ExecBase        =       4
OpenLib         =       -552
CloseLib        =       -414
InstallInp      =       -30
RemoveInp       =       -36
WriteChar       =       -48
WriteStr        =       -72
ReadStr         =       -60

Start:  move.l  ExecBase,a6
        lea     CIOName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)     ; conio.library öffnen
        move.l  d0,CIOBase

        move.l  CIOBase,a6
        lea     WindowArgs,a0
        jsr     InstallInp(a6)  ; Input installieren
        move.l  d0,InpUnit
        move.l  d0,a5

        lea     HalloTxt,a0
        jsr     WriteStr(a6)    ; "HalloText" ausgeben

Loop:   move.l  #">",d0         ; ">" Prompt ausgeben
        jsr     WriteChar(a6)

        lea     Puffer,a0       ; Zeichenkette einlesen
        jsr     ReadStr(a6)

        cmp.l   #"exit",Puffer  ; Eingabe = "exit"?
        bne     Loop            ; Nein, dann wiederholen

        move.l  a5,a0
        jsr     RemoveInp(a6)   ; Input entfernen

        move.l  ExecBase,a6
        move.l  CIOBase,a1
        jsr     CloseLib(a6)    ; Library schließen
```

```
      rts

CIOName:      dc.b    "conio.library",0
              even
WinName:      dc.b    "exit - beendet das Programm !",0
              even
Puffer:
HalloTxt:     dc.b    "Dies ist ein Demonstrationsprogramm für die "
              dc.b    "conio.library!",10,10
              dc.b    "<Bitte geben sie `exit` ein>",10,10,0
              even

CIOBase:      dc.l    0
InpUnit:      dc.l    0
WindowArgs: dc.w      0,0,640,200
              dc.b    1,3
              dc.l    $200,$11002,0,0,WinName,0,0
              dc.w    100,50,200,100,1
```

Programm 10.5: Benutzung der ConIO-Library

10.6 Das Narrator-Device

Bei dem Narrator-Device handelt es sich um ein "Gerät", welches die sprachliche Ausgabe von Texten ermöglicht. Dabei muß der auszugebende Text zuvor mit Hilfe der Translator-Library in Lautschrift übersetzt werden. Die Funktion, die man dazu benutzt, heißt Translate und benötigt vier Parameter.

Translate	=	**-30 (Translator-Library)**

inputString	**a0**	<	Zeiger auf Text, der konvertiert werden soll.
outputBuffer	**a1**	<	Zeiger auf einen Speicherbereich, in dem die Lautschrift gespeichert werden soll.
inputLength	**d0**	<	Länge des zu übersetzenden Textes.
bufferSize	**d1**	<	Größe des Puffers, in dem die Lautschrift gespeichert werden soll.

Erklärung Durch die Funktion Translate wird der angegebene Text in Lautschrift übersetzt, welche in den angegebenen Speicherbereich abgelegt wird.

Als nächstes müssen wir uns die beiden IORequest-Strukturen ansehen, die speziell für das Narrator-Device angelegt worden sind.

narrator_rb:

```
00      ds.b    IOStdReq,48         ; Standard IORequest-Struktur
48      dc.w    rate                ; Worte pro Minute
50      dc.w    pitch               ; Frequenz in Hz
52      dc.w    mode                ; Betonungsmodus
54      dc.w    sex                 ; Geschlecht der Stimme
56      dc.l   *ch_masks            ; Zeiger auf Channel-Masks
60      dc.w    nm_masks            ; Anzahl der Masken
62      dc.w    volume              ; Lautstärke
64      dc.w    sampfreq            ; Abtastrate
66      dc.b    mouths              ; Mund-Flag
67      dc.b    chanmask            ;┐ Kanalmasken
68      dc.b    numchan             ;┘ (intern benutzt)
69      dc.b    pad                 ; Füllbyte
70              SIZEOF
```

IOStdReq
Normale IOStdRequest-Struktur, in welche die normalen Parameter eingetragen werden.

rate
Rate bestimmt die Sprechgeschwindigkeit, gemessen in Worten pro Minute.

```
MINRATE     =     40
MAXRATE     =     400
```

pitch
Durch Pitch wird die Basisfrequenz (Hz) angegeben.

```
MINPITCH    =     65
MAXPITCH    =     320
```

mode
Mit dem Wert von mode kann der Betonungsmodus eingestellt werden.

```
NATURALF0   =     0
ROBOTICF0   =     1
```

sex
Geschlecht des Sprechers.

```
MALE        =     0
FEMALE      =     1
```

ch_masks
Zeiger auf Audio-Kanal-Masken.

nm_masks
Größe des Kanaldaten-Feldes.

volume
Lautstärke der Sprachausgabe.

MINVOL	=	0
MAXVOL	=	64

sampfreq
Abtastrate.

MINFREQ	=	5000
MAXFREQ	=	28000

mouths
Boolscher Ausdruck, der angibt, ob das Device Munddaten generieren soll (0 = keine Daten).

chanmask, numchan
Intern benutzte Datenwerte, die über die Kanalmaskendaten Auskunft geben.

pad
Füllbyte, um auf gerade Adresse zu kommen.

Wenn man Munddaten vom Device lesen will, muß man die folgende Struktur benutzen.

mouth_rb:

```
00      ds.b    narrator_rb         ; normale Narr-IOReq-Struktur
70      dc.b    width               ; Breite
71      dc.b    height              ; Höhe
72      dc.b    shape               ; Höhe/Breite
73      dc.b    pad                 ; Füllbyte
74              SIZEOF
```

narrator_rb
Normale Narrator-IORequest-Struktur.

width, height
Breite und Höhe.

shape
Kompression (Höhe/Breite)

pad
Füllbyte

Sollte ein Fehler bei der Ausgabe aufgetreten sein, so bekommt man einen der folgenden Fehlerwerte übergeben.

Name	Wert	Bedeutung
ND_NotUsed	-01	nicht benutzt
ND_NoMem	-02	Speicher konnte nicht belegt werden
ND_NoAudLib	-03	Audio-Device konnte nicht geöffnet werden
ND_MakeBad	-04	Fehler beim Erstellen der Library

```
ND_UnitErr          -05        angegebene Unit != 0
ND_CantAlloc        -06        Audiokanäle konnten nicht belegt werden
ND_Unimpl           -07        falsches Kommando
ND_NoWrite          -08        "mouth shape" gelesen ohne zu schreiben
ND_Expunged         -09        Fehler beim Öffnen
ND_PhonErr          -20        Fehler bei Aussprache der Lautschrift
ND_RateErr          -21        Fehler durch rate-Wert
ND_Pitch            -22        Fehler durch pitch-Wert
ND_Sex              -23        Fehler durch sex-Einstellung
ND_ModeErr          -24        Fehler durch mode-Einstellung
ND_FreqErr          -25        Fehler durch sampfreq-Wert
ND_VolErr           -26        Fehler durch volume-Wert
```

Zum Abschluß wollen wir uns noch das Demonstrationsprogramm ansehen, welches den Umgang mit dem Device eindeutig zeigen soll.

```
*
* Kapitel 10
* Demonstrationsprogramm für das Narrator-Device
*

ExecBase      =       4
OpenLib       =       -552
CloseLib      =       -414
DoIO          =       -456
Translate     =       -30
DeleteIOFast  =       -60
CreateIOFast  =       -54

Start:
        move.l  ExecBase,a6
        lea     TraName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)     ; Translator-Library öffnen
        move.l  d0,TraBase
        beq     TraError

        lea     IOName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)     ; IO-Library öffnen
        move.l  d0,IOBase
        beq     IOError

        move.l  IOBase,a6
        lea     NarrDevice,a0
        moveq   #0,d0
        moveq   #0,d1
        move.l  #70,d2
        jsr     CreateIOFast(a6)        ; IOReq-Fast erstellen
        move.l  d0,IORequest
```

```
        bsr     Speak           ; sprechen

        move.l  IOBase,a6
        move.l  IORequest,a0
        jsr     DeleteIOFast(a6)      ; IOReq-Fast löschen

        move.l  ExecBase,a6
        move.l  IOBase,a1
        jsr     CloseLib(a6)    ; IO-Library schließen

IOError:
        move.l  TraBase,a1
        jsr     CloseLib(a6)    ; Translator-Library schließen

TraError:
        rts

Speak:  lea     Puffer,a0       ; Adresse des Textes
                lea     TransMem,a1     ; Adresse des Puffers
        move.l  #PEnd-Puffer,d0       ; Länge des Textes
        move.l  #512,d1         ; Größe des Puffers
        move.l  TraBase,a6      ; Translator-Base nach a6
        jsr     Translate(a6)   ; übersetzen

        move.l  IORequest,a0    ; Zeiger auf IORequest nach a0
        move.w  #150,48(a0)     ; rate = 150
        move.w  #110,50(a0)     ; pitch   = 110
        move.w  #0,52(a0)       ; mode = 0
        move.w  #0,54(a0)       ; sex  = 0
        move.l  #MaskData,56(a0)      ; mask          = #MaskData
        move.w  #4,60(a0)       ; nummask = 4
        move.w  #64,62(a0)      ; volume   = 64
        move.w  #22200,64(a0)   ; sampfreq  = 222000

        move.l  IORequest,a1    ;
        move.w  #3,28(a1)       ; io_Command = schreiben
        move.l  #512,36(a1)     ; io_Length = Pufferlänge
        move.l  #TransMem,40(a1)      ; io_Data = Zeiger auf Daten
        move.l  ExecBase,a6
        jsr     DoIO(a6)        ; sprechen

        rts

* Datenbereich

IOName:         dc.b    "io.library",0
                even
NarrDevice:     dc.b    "narrator.device",0
                even
TraName:        dc.b    "translator.library",0
                even
InpUnit:        dc.l    0
TraBase:        dc.l    0
IORequest:      dc.l    0
IOBase:                 dc.l            0
```

```
MaskData:       dc.b    3,5,10,12

TransMem:       ds.b    512

Puffer:         dc.b    "hello world ",0
PEnd:
                even
```

Programm 10.6: Demonstration zum Narrator-Device

Kapitel 11
Konstruktion eines eigenen Devices

Initialisierung des Devices

Erweiterung der Lib-Standardroutinen

Device-Routinen und Verwaltungs-Task

Funktionen des Device

Demonstration und Quelltext

Nachdem wir uns einige Devices angesehen und sie benutzt haben, wollen wir uns nun damit beschäftigen, ein Device selbst zu programmieren. Dabei können wir die im Kapitel 9 erstellte Library als Basis benutzen, da ein Device zunächst aus einer ganz normalen Library besteht.

Auch der Lade- und Initialisierungsvorgang ist mit den Libraries identisch. So wird das Device mit LoadSeg in den Speicher geladen und dann mit InitResident eingerichtet. Dabei erhält die Initialisierungsroutine den Zeiger auf die Segment-Liste, die wir, wie bei der Library, zwischen-speichern müssen. **Neu** im Gegensatz zu den Libraries ist, daß die Initialisierungsroutine einen Task mit zugehörigem Message-Port einrichtet. Dieser Task soll dann die gewünschten Operationen mit dem "Gerät" durchführen. Um mit dem Task in Verbindung zu treten, stellt das Device neben den vier Library-Standardfunktionen (Close, Open, Expunge, ExtFunc) noch zwei weitere Device-Standardfunktionen zur Verfügung (BeginIO, AbortIO). Dabei wird durch die BeginIO-Funktion eine angegebene IORequest-Struktur an den Device-Task-Message-Port gesendet. Mit AbortIO kann die Bearbeitung eines bestimmten IORequests abgebrochen werden.

11.1 Initialisierung des Devices

Das soll erst einmal als Übersicht reichen. Anfangen wollen wir mit der Initialisierungsroutine. Ausgehend von der LibInitRoutine brauchen wir nur die folgenden Einträge zu ergänzen.

Zunächst müssen wir Speicher für den Stack des Tasks belegen und die Adresse in die Task-Struktur eintragen.

```
        ...
        move.l      #MsgPort,38(a5)     ; Zeiger auf MsgPort
                                        ; eintragen
        move.l      ExecBase,a6         ; Stackspeicher belegen
        move.l      #600,d0             ; 600 Bytes
        moveq       #0,d1               ; egal von welchem Typ
        jsr         AllocMem(a6)
        move.l      d0,tc_SPLower       ; Stackuntergrenze eintragen
        add.l       #600,d0
        move.l      d0,tc_SPUpper       ; Stackobergrenze eintragen
        move.l      d0,tc_SPReg         ; Stackzeiger setzen
        ...
```

Bild 11.1: Stack-Speicher belegen

Nachdem wir den Stack belegt und die letzten Eintragungen in die Task-Struktur erledigt haben, können wir nun den Task einrichten.

```
...
        move.l    #TaskStruktur,a1     ; Zeiger auf einzurichtende Task-
        move.l    a1,42(a5)            ; Struktur nach a1 und zusätzlich
                                       ; in den Datenbereich eintragen.
        move.l    #DevTask,a2          ; Zeiger auf Anfangsadresse
        sub.l     a3,a3                ; keine eigene Schlußroutine
        jsr       AddTask(a6)          ; Task einrichten lassen
        ...
```

Bild 11.2: Einrichtung des Tasks

Haben wir den Task eingerichtet, müssen wir die Register wieder restaurieren und können dann die DevInitRoutine verlassen.

11.2 Erweiterung der Lib-Standardroutinen

Neben der DevInitRoutine müssen auch noch die Close- und die Expunge-Routine erweitert werden. Die Close-Routine muß lediglich um zwei weitere Einträge ergänzt werden. Durch sie wird die angegebene IORequest-Struktur unbrauchbar gemacht.

```
        ...
        move.l    #-1,20(a1)           ; io_Device und io_Unit Eintrag der
        move.l    #-1,24(a1)           ; IORequest-Struktur löschen (-1)
        ...
```

Bild 11.3: Erweiterung der Close-Routine

Bei der nun folgenden Expunge-Funktion sehen wir uns nur die neu hinzugekommenen Teile an. Dabei muß der Task entfernt und der Stackbereich wieder freigegeben werden. Der Stackbereich hätte natürlich auch mit der AllocEntry-Funktion belegt werden können. Hätte man dann noch die Struktur in die MemList des Tasks eingetragen, so wäre automatisch der Stackbereich beim Entfernen des Tasks freigegeben worden.

```
        ...
        lea       TaskStruktur,a1      ; Task "ausschalten"
        jsr       RemTask(a6)

        move.l    tc_SPLower,a1        ; Zeiger auf Anfang des Bereichs
        move.l    #600,d0              ; Größe des Stacks
        jsr       FreeMem(a6)          ; Stackspeicher freigeben
        ...
```

Bild 11.4: Erweiterung der Expunge-Routine

11.3 Device-Routinen und Verwaltungs-Task

Neben den vier Standardfunktionen benötigt ein Device noch zwei Spezialfunktionen. Wie wir wissen, wird die BeginIO-Funktion von den Exec-Funktionen DoIO und SendIO aufgerufen, damit sie die Nachricht an den Device-Task schickt. Natürlich brauchen wir auch für unseren Task eine solche Funktion.

```
BeginIO:
        movem.l     d1-d6/a0-a6,-(a7)

        clr.b       31(a1)              ; io_Error löschen
        bclr        #0,30(a1)           ; Quick Bit löschen

        move.l      38(a6),a0           ; Zeiger auf MsgPort auslesen
        move.l      ExecBase,a6
        jsr         PutMsg(a6)          ; Nachricht an Task schicken

        moveq       #0,d0

        movem.l     (a7)+,d1-d6/a0-a6
        rts
```

Bild 11.5: Die BeginIO-Routine

Zuerst wird der io_Error-Eintrag der TestIORequest-Struktur und das Quick-Bit im io_Flags-Eintrag gelöscht. Danach wird die Nachricht an den Message-Port des Device-Tasks geschickt. Das Quick-Bit wird von der Funktion DoIO gesetzt, bevor die Device-Routine BeginIO abgearbeitet wird. Nachdem BeginIO beendet worden ist, wird je nachdem, ob das Bit gesetzt ist oder nicht, auf die Bearbeitung gewartet (Quick-Bit gesetzt, dann nicht warten).

Die zweite Standardfunktion für Devices (AbortIO) wird von unserem Test-Device nicht unterstützt. Hier könnte man z.B. die übergebene TestIORequest-Struktur aus der Liste des Message-Ports entnehmen.

Jetzt kommen wir zum "Herz" des Devices, dem Device-Task. Er muß die am Message-Port ankommenden Nachrichten auswerten und die angeforderte Operation ausführen.

Wenn keine Nachricht am Port anliegt, wartet der Task solange, bis der Message-Port eine Nachricht empfangen hat. Dann wird sie mittels GetMsg aus der Liste entnommen.

```
DevTask:
        move.l      LibPtr,a5           ; Zeiger auf Basisadresse der Lib
                                        ; nach a5
```

```
MessageLoop:
        move.l      ExecBase,a6
        move.l      38(a5),a0           ; Zeiger auf Port nach a0 und
        jsr         WaitPort(a6)        ; auf Nachrichten warten
        move.l      d0,46(a5)           ; Nachricht zwischenspeichern

        lea         MsgPort,a0
        jsr         GetMsg(a6)          ; Nachricht entfernen
        tst.l       46(a5)              ; "falscher Alarm"?
        beq         MessageLoop
        ...
```

Bild 11.6: Message-Warteschleife

Nachdem eine Nachricht vom Message-Prot abgeholt worden ist, kann sie im nächsten Schritt bearbeitet werden. Dabei benutzen wir eine Offsettabelle, über deren Werte wir die gewünschte Funktion anspringen.

```
; Nachrichtenbehandlung

        move.l      46(a5),a4           ; Zeiger auf IORequest-Block
        moveq       #0,d0               ; d0 löschen
        move.w      28(a4),d0           ; io_Command auslesen
        lsl.l       #1,d0               ; * 2
        lea         StartCMD,a0         ; Zeiger auf Tabelle mit Offsets
        move.w      (a0,d0),d0          ; Offset auslesen
        jsr         (a0,d0)             ; Routine aufrufen

        move.l      46(a5),a1
        jsr         ReplyMsg(a6)        ; Nachricht bestätigen
        bra         MessageLoop


StartCMD:
        dc.w        CMDInvalid-StartCMD     ; 0
        dc.w        CMDReset-StartCMD       ; 1
        dc.w        CMDRead-StartCMD        ; 2
        dc.w        CMDWrite-StartCMD       ; 3
        dc.w        CMDUpDate-StartCMD      ; 4
        dc.w        CMDClear-StartCMD       ; 5
        dc.w        CMDStop-StartCMD        ; 6
        dc.w        CMDStart-StartCMD       ; 7
        dc.w        CMDAbort-StartCMD       ; 8

        dc.w        CMDExecuteSub-StartCMD  ; 9
```

Bild 11.7: Nachrichtenauswertung

Bevor man die nächste Nachricht vom Port abholt, muß man die bearbeitete mit ReplyMsg wieder zurücksenden.

11.4 Funktionen des Device

Nachdem wir alle Standardroutinen unseres Devices angesehen haben, fehlen uns noch die eigentlichen Funktionen. Da es sich hier, wie der Name schon sagt, nur um ein Test-Device handelt, ist die Anzahl der Funktionen nicht besonders groß. Es ist lediglich eine, die aber den Vorteil eines Devices deutlich macht. Die Funktion heißt ExeSub und hat die Kommandonummer (CMDExeSub =) 9. Die Funktion ist eigentlich recht einfach. Sie liest aus der gesendeten TestIORequest-Struktur die Adresse einer Subroutine aus und arbeitet sie ab. Dadurch ist es möglich, daß ein Task eine bestimmte Aufgabe vom Device erledigen läßt. Die Funktion ist lediglich als kleine Demonstration gedacht und sollte nicht weiter benutzt werden!

```
CMDExecuteSub:                          ; ExeSub-Funktion
        move.l     46(a5),a0            ; Zeiger auf Nachricht auslesen
        move.l     32(a0),a0            ; Adresse der Routine nach a0
        movem.l    a0-a6/d0-d7,-(a7)    ; Register retten
        jsr        (a0)                 ; Routine anspringen
        movem.l    (a7)+,a0-a6/d0-d7    ; Register restaurieren
        rts
```

Bild 11.8: Die Device-Funktion ExecuteSub

Die zum Aufrufen benutzte TestIORequest-Struktur hat folgenden Aufbau:

TestIORequest-Struktur:

```
00      dc.l    io_Succ          ;┐
04      dc.l    io_Pred          ;│
08      dc.b    io_Type          ;│
09      dc.b    io_Pri           ;│
10      dc.b    io_Name          ;│
14      dc.l    io_ReplyPort     ;│ IORequest-Struktur
18      dc.w    io_Length        ;│
20      dc.l    io_Device        ;│
24      dc.l    io_Unit          ;│
28      dc.w    io_Command       ;│
30      dc.b    io_Flags         ;│
31      dc.b    io_Error         ;┘

32      dc.l    io_SubAddress    ; Adresse der Subroutine
36              io_SIZEOF
```

io_SubAddress
Der Eintrag io_SubAddress muß die Adresse einer Unterroutine enthalten, die von dem Device ausgeführt werden soll.

Wichtig ist vielleicht noch der Aufbau der Device-(Library)-Struktur mit dem zugehörigen Datenteil:

TestDevice-Struktur:

```
00      ds.l    34              ; Library-Struktur
34      dc.l    0               ; Zeiger auf Segment-Liste
38      dc.l    0               ; Zeiger auf MsgPort
42      dc.l    0               ; Zeiger auf Task-Struktur
46      dc.l    0               ; Zeiger auf empfangenen IORequest
```

11.5 Demonstration und Quelltext

Wie man nun unser Device benutzt, zeigt das Demonstrationsprogramm, welches sich auf der Programm-Diskette befindet (PRG_11_1.S). Es öffnet unser Device mit Hilfe der CreatIO-Fast-Funktion und trägt die Adresse einer Test-Routine ein, welche die Hintergrundfarbe verändert. Dann wird die IORequest-Struktur abgeschickt und durch den Task solange eine Nachricht auf dem CLI-Fenster ausgegeben, bis die IORequest-Struktur abgearbeitet worden ist. Damit das Programm funktionieren kann, benötigt es natürlich das nun folgende Device als ausführbare Datei im Verzeichnis "devs:".

```
*
* Kapitel 11
* Quelltext des test.device
*

ExecBase        =       4
Output          =       -60
Write           =       -48
OpenLib         =       -552
CloseLib        =       -414
Remove          =       -252
FreeMem         =       -210
AllocMem        =       -198
WaitPort        =       -384
GetMsg          =       -372
PutMsg          =       -366
ReplyMsg        =       -378
AddTask         =       -282
RemTask         =       -288

Start:
        move.l  ExecBase,a6     ; Dos-Library öffnen
        lea     DosName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)
```

```
        move.l  d0,a6
        beq     DosError

        jsr     Output(a6)      ; Ausgabekanal ermitteln
        move.l  d0,d1

        move.l  #TextA,d2       ; Text ausgeben
        move.l  #TextE-TextA,d3
        jsr     Write(a6)

        move.l  a6,a1                   ; DosLibrary schließen
        move.l  ExecBase,a6
        jsr     CloseLib(a6)

DosError:
        moveq  #0,d0
        rts

TextA:          dc.b    10,"> TestDevice Version 0.1 <",10,10
TextE:          even

; Nun folgt der eigentliche Teil des Devices

LibResident:    dc.w    $4AFC           ; rt_MatchWord
                dc.l    LibResident     ; rt_MatchTag
                dc.l    EndResident     ; rt_EndSkip
                dc.b    %10000000       ; rt_Flags
                dc.b    1               ; rt_Version
                dc.b    3               ; rt_Type
                dc.b    0               ; rt_Pri
                dc.l    DevName         ; rt_Name
                dc.l    LibIDString     ; rt_IDString
                dc.l    LibInitData     ; rt_Init

DevName:        dc.b    "test.device",0
                even
LibIDString:    dc.b    "TestDevice v0.1",0
                even

LibInitData:    dc.l    50              ; LibSize (34) + SegList (4)
                                        ; MsgPort (4) + TaskPtr (4)
                                        ; Message (4) = 50
                dc.l    FuncTab
                dc.l    DataTab
                dc.l    LibInitRout

FuncTab:        dc.l    Open            ;
                dc.l    Close           ; Library-Standardfunktionen
                dc.l    Expunge         ;
                dc.l    ExtFunc         ;

                dc.l    BeginIO         ; Device-Standardfunktionen
                dc.l    AbortIO         ;
                dc.l    -1
```

```
DataTab:        dc.b    %11100000,0    ; Datentabelle für InitStruct
                dc.w    8
                dc.b    3,0

                dc.b    %11000000,0
                dc.w    10
                dc.l    DevName

                dc.b    %11100000,0
                dc.w    14
                dc.b    6,0

                dc.b    %11000000,0
                dc.w    20
                dc.w    3,4

                dc.l    0

DosName:        dc.b "dos.library",0
                even

; Initialisierungsroutine:

LibInitRout:
        movem.l  d0-d6/a0-a6,-(a7)

        move.l   d0,a5              ; Library-Basis nach a5
        move.l   d0,LibPtr
        move.l   a0,34(a5)          ; Segment-Liste eintragen
        move.l   #MsgPort,38(a5)          ; Zeiger auf MsgPort
                                    ; eintragen
        move.l   ExecBase,a6        ; Stackspeicher belegen
        move.l   #600,d0            ; 600 Bytes
        moveq    #0,d1
        jsr      AllocMem(a6)
        move.l   d0,tc_SPLower      ; Stackpointer eintragen
        add.l    #600,d0
        move.l   d0,tc_SPUpper
        move.l   d0,tc_SPReg

        move.l   #TaskStruktur,a1         ; Zeiger auf Task-Struktur
        move.l   a1,42(a5)          ; eintragen in Datenbereich
        move.l   #DevTask,a2        ; Anfang der Bearbeitung
        sub.l    a3,a3
        jsr      AddTask(a6)        ; Device-Task eintragen

        move.l   LibPtr,d0

LibInitEnd:
        movem.l  (a7)+,d0-d6/a0-a6
        rts

; Open-Routine

Open:
        bclr     #3,14(a6)          ; ExpungBit = 0
```

```
        addq.w   #1,32(a6)       ; OpenCnt ++
        move.l   a6,d0
        rts

; Close-Routine

Close:  movem.l  a2-a3/a5,-(a7)
        moveq    #0,d0           ; Segment-List = 0
        move.l   #-1,20(a1)      ; io_Device löschen
        move.l   #-1,24(a1)      ; io_Unit löschen

        subq.w   #1,32(a6)       ; OpenCnt --
        bne      CloseEnd

        btst     #3,14(a6)       ; Device entfernen ?
        beq      CloseEnd

        bsr      Expunge         ; entfernen

CloseEnd:
        movem.l  (a7)+,a2-a3/a5
        rts

; Expunge-Routine

Expunge:
        movem.l  d2/a5-a6,-(a7)

        tst.w    32(a6)          ; kein Benutzer mehr ?
        beq      ExpungeBranch

        moveq    #0,d0           ; SegmentList = 0
        bset     #3,14(a6)       ; ExpungeBit setzen
        bra      ExpungeEnd      ; jetzt noch nicht entfernen

ExpungeBranch:
        move.l   a6,a5           ; Device aus Device-Liste entfernen
        move.l   ExecBase,a6
        move.l   a5,a1
        jsr      Remove(a6)      ; Remove

        lea      TaskStruktur,a1        ; Task "ausschalten"
        jsr      RemTask(a6)

        move.l   tc_SPLower,a1
        move.l   #600,d0
        jsr      FreeMem(a6)     ; Stackspeicher freigeben

        move.l   34(a5),d2       ; Segment-Listen-Ptr retten

        moveq    #0,d0
        move.w   16(a5),d0
        move.l   a5,a1
        sub.l    d0,a1
        add.w    18(a5),d0       ; Library-Struktur freigeben
        jsr      FreeMem(a6)     ; FreeMem
```

```
        move.l   d2,d0           ; Segment-Listen-Ptr zurückgeben

ExpungeEnd:
        movem.l  (a7)+,d2/a5-a6
        rts

; ExtFunc-Routine

ExtFunc:
        moveq    #0,d0           ; Rückgabewert = Null
        rts

; BeginIO-Routine

BeginIO:
        movem.l  d1-d6/a0-a6,-(a7)

        clr.b    31(a1)          ; io_Error löschen
        bclr     #0,30(a1)       ; Quick Bit löschen

        move.l   38(a6),a0       ; Zeiger auf MsgPort auslesen
        move.l   ExecBase,a6
        jsr      PutMsg(a6)      ; Nachricht an Task schicken

        moveq    #0,d0

        movem.l  (a7)+,d1-d6/a0-a6
        rts

; AbortIO-Routine

AbortIO:
        moveq    #0,d0
        rts

*
* *** Device-Task ***********************************
*

DevTask:
        move.l   LibPtr,a5       ; Zeiger auf Basisadresse der Lib
                                 ; nach a5

MessageLoop:
        move.l   ExecBase,a6
        move.l   38(a5),a0       ; Zeiger auf Port nach a0 und
        jsr      WaitPort(a6)    ; auf Nachrichten warten
        move.l   d0,46(a5)       ; Nachricht zwischenspeichern

        lea      MsgPort,a0
        jsr      GetMsg(a6)      ; Nachricht entfernen
        tst.l    46(a5)          ; "falscher Alarm"?
        beq      MessageLoop
```

```
; Nachrichtenbehandlung

        move.l   46(a5),a4        ; Zeiger auf IORequest-Block
        moveq    #0,d0            ; d0 löschen
        move.w   28(a4),d0        ; io_Command auslesen
        lsl.l    #1,d0            ; * 2
        lea      StartCMD,a0      ; Zeiger auf Tabelle mit Offsets
        move.w   (a0,d0),d0       ; Offset auslesen
        jsr      (a0,d0)          ; Routine aufrufen

        move.l   46(a5),a1
        jsr      ReplyMsg(a6)     ; Nachricht bestätigen
        bra      MessageLoop


StartCMD:       dc.w    CMDInvalid-StartCMD ; 0
                dc.w    CMDReset-StartCMD   ; 1
                dc.w    CMDRead-StartCMD    ; 2
                dc.w    CMDWrite-StartCMD   ; 3
                dc.w    CMDUpDate-StartCMD  ; 4
                dc.w    CMDClear-StartCMD   ; 5
                dc.w    CMDStop-StartCMD    ; 6
                dc.w    CMDStart-StartCMD   ; 7
                dc.w    CMDAbort-StartCMD   ; 8

                dc.w    CMDExecuteSub-StartCMD              ; 9

CMDInvalid:     ; Die neun Standard-Device-Funktionen werden
CMDReset:       ; von unserem Device nicht unterstützt. Deshalb
CMDRead:        ; werden sie lediglich mit einem "rts" quittiert.
CMDWrite:
CMDUpDate:
CMDClear:
CMDStop:
CMDStart:
CMDAbort:
        rts      ; < Hier das besagte "RTS"

; Die nun folgende ExeSub-Funktion dient nur als
; Beispiel und sollte nicht weiter verwendet werden !

CMDExecuteSub:                    ; ExeSub-Funktion
        move.l   46(a5),a0        ; Zeiger auf Nachricht auslesen
        move.l   32(a0),a0        ; Adresse der Routine nach a0
        movem.l  a0-a6/d0-d7,-(a7)      ; Register retten
        jsr      (a0)             ; Routine anspringen
        movem.l  (a7)+,a0-a6/d0-d7      ; Register restaurieren
        rts

* Datenbereich

LibPtr:         dc.l    0               ; Speicherbereich für Base-Pointer

MsgPort:        dc.l    0,0             ; Message-Port-Struktur
                dc.b    4,0
                dc.l    DevName
```

```
                dc.b    0,24
                dc.l    TaskStruktur

; Es ist unbedingt erforderlich, den Listenkopf der
; ankommenden Nachrichten zu initialisieren!

mp_Head:        dc.l    mp_Tail         ; Zeiger auf nächste Nachricht
mp_Tail:        dc.l    0               ; Null-Kennung
mp_TailPred:
                dc.l    mp_Head         ; Zeiger auf vorangegangene Nach.
                dc.b    0,0

TaskStruktur:
                dc.l    0,0
                dc.b    1,0             ; Node-Struktur
                dc.l    DevName         ;

                dc.b    0               ; tc_Flags
                dc.b    0               ; tc_State
                dc.b    0               ; tc_IDNestCnt
                dc.b    0               ; tc_TDNestCnt
                dc.l    0               ; tc_SigAlloc
                dc.l    0               ; tc_SigWait
                dc.l    0               ; tc_SigRecvd
                dc.l    0               ; tc_SigExcept
                dc.w    0               ; tc_TtrapAlloc
                dc.w    0               ; tc_TrapAble
                dc.l    0               ; tc_ExceptData
                dc.l    0               ; tc_ExceptCode
                dc.l    0               ; tc_TrapData
                dc.l    0               ; tc_TrapCode
tc_SPReg:       dc.l    0               ; tc_SPReg
tc_SPLower:     dc.l    0               ; tc_SPLower
tc_SPUpper:     dc.l    0               ; tc_SPUpper
                dc.l    0               ; tc_Switch
                dc.l    0               ; tc_Launch
                dcb.b   14              ; tc_MemEntry
                dc.l    0               ; tc_UserData

EndResident:
```

Programm "test.device.s" (Quellcode des "Test-Device")

Zum Schluß sei noch erwähnt, daß sich die Anzahl der Funktionen, die ein Device hat, nicht nur auf sechs (Library-Standardfunktionen + AbortIO + BeginIO) beschränken muß, da man das Device auch als Library benutzen kann. Außerdem ist es möglich, wenn mehrere Geräte benutzt werden, auch mehrere Tasks zu erstellen. So legt das TDD für jedes Laufwerk einen eigenen Task an.

Kapitel 12
DOS für Fortgeschrittene

Der Aufbau einer DOS-Diskette

Berechung von Blockchecksummen

Der Bootblock

Die Basisstruktur der DOS-Library

Bei der Besprechung der DOS-Library im 4. Kapitel haben wir ein paar Features, die zum DOS gehören, aber "damals" noch zu kompliziert waren, weggelassen. Nun verfügen wir über genug Systemwissen, um auch das letzte Eckchen des DOS verstehen zu können.

Wir beginnen mit dem Aufbau einer DOS-Diskette, wobei wir lernen werden, wie das DOS Daten auf seinen Speichermedien verwaltet. Dann behandeln wir die Basisstruktur der DOS-Library und diversen anderen wichtigen Strukturen. Schließlich wollen wir uns mit den DOS-Packets, der Parameterübergabe, den ".info"-Dateien und den "executable"-Dateien beschäftigen.

12.1 Der Aufbau einer DOS-Diskette

Der Begriff "Diskette" bezieht sich wieder einmal nicht nur auf Floppy-Disks, sondern auch auf Festplatten, die RAM-Disk "RAD:" und sonstige "Floppy-ähnliche" Geräte. Allerdings gibt es gewisse Unterschiede zwischen den einzelnen Geräten, worauf wir noch näher eingehen werden. Wenn Sie also im folgenden "Diskette" oder "Disk" lesen, ist, falls nichts anderes gesagt wird, irgendein floppy-ähnliches Medium gemeint.

Die Einrichtung im Amiga-System, die für die Verwaltung der Datenträger zuständig ist, nennt sich "File-System" (Dateisystem). Die meisten Funktionen der DOS-Library reichen die, ihnen aufgetragene Arbeit eigentlich nur an das Filesystem weiter und warten, bis es die Arbeit erledigt hat (recht faule Library). Bei der Betrachtung des Diskettenaufbaus werden wir also vom Filesystem, und nicht vom DOS sprechen.

Für den Umgang mit Disketten kennt der Amiga zwei Filesysteme: das alte FileSystem, genannt OldFileSystem (OFS) und das FastFileSystem (FFS). Letzteres ist eine Weiterentwicklung des alten Filesystems. Ab Kickstart 2.0 ist es auch für Floppy-Disketten benutzbar, vorher war es nur für Festplatten reserviert. Wir werden uns zunächst mit dem alten Filesystem befassen, auf die Unterschiede zum FastFileSystem kommen wir dann ganz automatisch.

Im Kapitel über das Trackdisk-Device haben wir schon erfahren, daß eine Diskette in Tracks, Seiten und Sektoren und damit in Blöcke aufgeteilt ist, von denen jeder 512 Bytes umfaßt. Das Amiga-Filesystem orientiert sich ebenfalls an dieser Block-Unterteilung. Neben den sog. "Datenblöcken", in die (unter anderem) die eigentlichen Daten geschrieben werden, kennt das Filesystem verschiedene andere Blocktypen, die allesamt der Verwaltung dienen.

Jeder Diskblock wird über seine Nummer angesprochen. Ein Rechenbeispiel: Eine Floppy-Disk hat 80 Tracks (0-79), 2 Seiten, und jeder Track hat 11 Sektoren (0-10). Daraus ergibt sich eine Blockzahl von 80*2*11 = 1760. Die Blöcke einer Floppy-Disk bekommen also die Nummern 0 bis 1759. Bei anderen Speichermedien, speziell bei Festplatten, dürfte die Blockanzahl erheblich höher liegen.

Die verschiedenen Blocktypen, die das Filesystem kennt, sind folgende:

Directory-Blöcke
Dieser Blocktyp dient, wie der Name schon sagt, der Verwaltung von Verzeichnissen. Er enthält Informationen über den Namen, den Schutzstatus, das Datum des letzten Schreibzugriffes auf das Verzeichnis und als Kernstück eine Liste mit Zeigern auf die Verwaltungsblöcke der Objekte (Dateien oder Unterverzeichnisse), die sich in dem Verzeichnis befinden.

Fileheader-Blöcke
Diese Blöcke sind für die Verwaltung der Dateien zuständig. Auch sie enthalten den Namen, Schutzstatus, Datum usw. der Datei. Als Kernstück ist hier eine Liste der zu dieser Datei gehörigen Datenblöcke zu finden.

Der Root-Block
Hierbei handelt es sich um einen speziellen Directory-Block. Er enthält das Hauptverzeichnis einer Diskette. Anstatt eines Verzeichnisnamens ist hier der Diskettenname zu finden.

Der Bitmap-Block
Der Begriff "Bitmap" ist nicht zu verwechseln mit der Grafik-Bitmap. Die Bezeichnung hat aber einen ähnlichen Ursprung: Im der Bitmap ist verzeichnet, welche Diskblöcke frei und welche belegt sind. Jedem Block wird dabei ein Bit in der Tabelle zugeordnet - daher also Bitmap.

Filelist-Blöcke
In einem Fileheader-Block können maximal 72 zu einer Datei gehörigen Datenblöcke verwaltet werden. Wenn eine Datei mehr als 72 Blöcke umfaßt, werden ein bzw. mehrere Filelist-Blöcke eingerichtet, in dem bzw. denen die restlichen Datenblöcke verwaltet werden.

Datenblöcke
Hier stehen die eigentlichen Daten. Neben diesen sind aber auch noch Verwaltungsinformationen wie ein Zeiger zurück zum Fileheader, die Nummer dieses Datenblocks in der Blockfolge der Datei, ein Zeiger auf den nächsten Datenblock usw. Diese Zusatzinformationen sind bei der Restauration defekter Diskettenstrukturen sehr nützlich.

Nachdem wir nun einen groben Überblick über die Blocktypen haben, wollen wir jeden Typ einzeln besprechen.

12.1.1 Der Boot-Block

Nanu, werden Sie jetzt vielleicht denken, dieser Blocktyp kam in der obigen Liste doch gar nicht vor. Das stimmt, denn er gehört eigentlich nicht zu den Blöcken, die das Filesystem verwaltet. Der Boot-Block (eigentlich müßte es heißen: die Boot-Blöcke) belegt immer die ersten zwei Blöcke einer DOS-Diskette (Track 0, Seite 0, Sektor 0 und Track 0, Seite 0, Sektor 1). Für das System hat er eine zweifache Bedeutung: Erstens enthalten die ersten vier Bytes in diesem Doppelblock die Kennung des Dateisystems. Bei einer Diskette, die mit dem alten Dateisystem formatiert wurde, steht dort das Hex-Langwort $444F5300. Das entspricht den ASCII-Codes für die Großbuchstaben "D", "O", "S" und einem Nullbyte. Erinnern Sie sich noch an die InfoData-Struktur aus dem DOS-Einsteiger-Kapitel? Da tauchte dieser Wert schon einmal als Kennzeichen für eine Standard-DOS-Disk auf! Jetzt wissen Sie auch, woher er stammt.

Eine FFS-Diskette trägt die Kennung $444F5301, also lediglich ein Eins-Byte anstatt eines Nullbytes am Ende der Kennung. Damit eine Diskette vom Filesystem akzeptiert wird, muß sie die DOS-0-Kennung (ab Kickstart 2.0 auch DOS-1 für FFS-Disk) im Bootblock tragen.

Die zweite Bedeutung des Bootblocks kommt beim Systemstart (eben beim Booten) zum Tragen: Der Bootblock enthält ein Maschinenprogramm, das beim Start von der Diskette ausgeführt wird, noch bevor der Workbench-Screen erscheint oder die Startup-Sequence an die Reihe kommt. Das Programm kann, wenn man ein Spiel oder sonstiges Programm vor sich hat, eine Laderoutine, ein Intro oder etwas Ähnliches sein. Jetzt verstehen Sie vielleicht auch, warum manche Disketten sofort mit dem Laden beginnen, ohne daß vorher der Workbench-Screen sichtbar wird.

Eine Diskette ist nur dann bootfähig (d.h. man kann das System nur dann von ihr starten), wenn im Bootblock ein korrektes Programm vorhanden ist (was das heißt, werden wir noch sehen). Wenn dies nicht der Fall ist, kann man zwar per Filesystem auf die Disk zugreifen, aber keinen Systemstart von ihr ausführen. Bei einer CLI-installierten Diskette erfüllt das Bootblock-Programm nur den Zweck, einen Zeiger auf die Resident-Struktur der DOS-Library (Exec-Routine FindResident) zu ermitteln und diesen dem System, als Zeichen für die Einsatzbereitschaft des DOS, zurückzumelden. Als "brave" DOS-User brauchen wir uns um den Bootblock nicht zu kümmern. Wir werden später auf seine Programmierung zurückkommen, jetzt wollen wir ihn erst einmal als System-gegeben hinnehmen.

Die Blocktypen, die im folgenden vorgestellt werden, sind alle langwort-orientiert aufgebaut. Die 512 Bytes eines Blocks werden in 128 Langworte aufgeteilt, die bestimmte Funktionen erfüllen, d.h. mit bestimmten Werten belegt sind. Diese Belegungen wollen wir uns nun anschauen.

12.1.2 Der Root-Block

Beim Rootblock handelt es sich um einen speziellen Directory-Block. Er ist immer an einer festen Stelle, nämlich in der "Mitte" des Datenträgers. Bei Floppy-Disks ist das Track 40, Seite 0, Sektor 0, also Block 880.

Den Aufbau dieses Blocks und auch der weiteren Blocktypen stellen wir in einer Tabelle vor. Links stehen die Offsets der Einträge, zuerst als Langwort- und dann als Byte-Offset. Dann folgt der Inhalt des Eintrags und schließlich eine kurze Erklärung. Eine ausführliche Erklärung folgt, falls nötig, nach der Tabelle.

Der Aufbau des Root-Blocks

Offsets Long	Byte	Eintrag	Bedeutung
0	0	2	Primärer Blocktyp: T.SHORT
1	4	0	---
2	8	0	---
3	12	HT Size	Größe der Hash-Tabelle (72)
4	16	0	---
5	20	Checksum	Checksumme des Blocks
6	24	Hash Table	Hash-Tabelle (Zeiger auf Blöcke
7	28	...	der Objekte im Verzeichnis)
77	308		
78	312	Bitmap Flag	Bitmap ok? (0=Nein, <>0=Ja)
79	316	Bitmap Pages	Zeiger auf Bitmap-Blöcke
...	...	...	
104	416		
105	420	Days	Datum und Uhrzeit des letzten
106	424	Mins	Schreibzugriffs (im DateStamp-
107	428	Ticks	Format)
108	432	Disk Name	Diskettenname (BCPL-String)
...	...	...	
120	480		
121	484	CreateDays	Datum und Uhrzeit der Formatierung
122	488	CreateMins	der Diskette (DateStamp-Format)
123	492	CreateTicks	
124	496	0	---
125	500	0	---
126	504	0	---

127	508	1	Sekundärer Blocktyp: ST.ROOT

Falls im Eintrag eine bestimmte Zahl steht, so ist sie als Konstante anzusehen, die bei diesem Blocktyp immer dort stehen muß. Variable Werte sind durch englische Texte (die von Commodore vorgegebenen) gekennzeichnet.

Der Aufbau der verschiedenen Blocktypen ist recht ähnlich. Daher gibt es Einträge, die nur in bestimmten Blocktypen eine Bedeutung haben, in anderen aber nicht. Das Zeichen "——" in der Erklärung besagt, daß dieser Eintrag in diesem Blocktyp keine Bedeutung hat und auf 0 stehen sollte.

Nun zu den einzelnen Einträgen des Rootblocks:

Blocktyp — Das erste und letzte Langwort in einem Verwaltungsblock gibt immer den Blocktyp an. Beim Rootblock steht im ersten Eintrag eine 2 und im zweiten eine 1, was für T.SHORT bzw. ST.ROOT steht.

HT Size / Hash Table — Das Amiga-Filesystem verwendet eine besondere Methode, um aus dem Namen einer gesuchten Datei oder eines Verzeichnisses die Position ihres/seines Verwaltungsblocks zu bestimmen. In diesem Zusammenhang spielt die Hash-Tabelle eine wichtige Rolle. Im Abschnitt über die Directory-Blöcke werden wir darauf zurückkommen.

Checksum — Über jeden Block berechnet das Filesystem grundsätzlich eine Checksumme, um etwaige Fehler schnell erkennen zu können. In allen Blocktypen bis auf den Bitmap-Block wird die Checksumme im 5. Langwort eingetragen. Auf die Art der Berechnung werden wir gleich kommen.

Bitmap Flag — Dieses Langwort gibt an, ob die Bitmap (siehe unten) gültig ist oder neu berechnet werden muß. Wenn hier eine 0 steht, ist die Bitmap ungültig, bei einem Wert ungleich 0 ist sie gültig.

Bitmap Pages — Die Bitmap eines Datenträgers kann aus mehreren Blöcken bestehen (bei einer Floppy-Disk reicht ein Block). Jeder dieser Blöcke wird Bitmap-Block genannt. In der Bitmap ist verzeichnet, welche Blöcke einer Diskette frei und welche belegt sind. Das ist für das Filesystem sehr wichtig zu wissen, damit es nicht schon belegte Blöcke noch einmal verwendet (Datenverlust). Mehr dazu im Abschnitt über den Bitmap-Block.

Days/Mins/Ticks — Hier wird das Datum und die Uhrzeit des letzten Schreibzugriffs auf die Diskette im DateStamp-Format (siehe DOS-Kapitel) abgelegt.

Disk Name — Das erste Byte dieses Eintrags gibt die Länge des Diskettennamens an, dann folgen die einzelnen Zeichen (BCPL-String).

CreateDays/Mins/Ticks Ebenfalls im DateStamp-Format wird der Zeitpunkt des Anlegens (Formatierens) der Diskette abgelegt.

Berechnung der Rootblock-Checksumme

Jeder Block bekommt vom Filesystem eine Checksumme. Auf diese Weise können fehlerhafte oder ungültige Blöcke schnell erkannt werden, da die Wahrscheinlichkeit, daß sich die eventuellen Datenfehler in einem Block genau so ausgleichen, daß die Checksumme wieder stimmt, sehr gering ist.

Das Amiga-Filesystem verwendet drei unterschiedliche Methoden der Checksummen-Berechnung, je nachdem, um welchen Blocktyp es sich handelt. Die Checksummen sind immer Langwörter. Für die Blocktypen Rootblock, Directory-Block, File-Header-Block, Data-Block und File-List-Block sieht die Berechnung folgendermaßen aus:

1. Lösche das Summen-Langwort und den eventuell vorhandenen bisherigen Checksummen-Eintrag (Langwort 5 im Block).
2. Subtrahiere alle Langwörter im Block nacheinander vom Summen-Langwort (Unterläufe werden nicht beachtet).
3. Das Summen-Langwort enthält dann die Checksumme.

Das Löschen der alten Checksumme ist notwendig, da die Checksumme selbst natürlich nicht in ihre eigene Berechnung eingehen darf. Die so berechnete Checksumme muß ins 5. Langwort eingetragen werden. In Assembler könnte die Berechnung so aussehen (in a0 wird der Start des Blocks im Speicher erwartet):

```
        clr.l   d0          ; Lösche Summen-Langwort
        clr.l   20(a0)      ; Lösche alte Checksumme
        move.l  a0,a1       ; Blockzeiger zur Bearbeitung nach a1
        moveq   #127,d1     ; Bearbeite 128 Langwörter
m1:     sub.l   (a1)+,d0    ; Nächstes Langwort subtrahieren
        dbra    d1,m1       ; Schleife
        move.l  d0,20(a0)   ; Berechnete Checksumme eintragen
```

Bild 12.1: Berechnung der Rootblock-Checksumme

12.1.3 Der Bitmap-Block

Die Bitmap dient, wie schon erwähnt, der Kennzeichnung von belegten und freien Blöcken. Als erstes die Aufbau-Tabelle:

Der Aufbau des Bitmap-Blocks

Offsets Long	Byte	Eintrag	Bedeutung
0	0	Checksum	Bitmap-Checksumme
1	4	Bitmap	Bitmap-Langwörter
2	8	...	
...	...	...	
127	508		

Der Bitmap-Block besteht also nur aus einer Checksumme und den eigentlichen Bitmap-Daten. Bevor wir letztere besprechen, zuerst die Vorgehensweise bei der Berechnung der Bitmap-Checksumme:

Berechnung der Bitmap-Checksumme

1. Lösche das Summen-Langwort
2. Subtrahiere die Langwörter 2-128 des Blocks vom Summen-Langwort (Unterläufe werden nicht beachtet).
3. Das Summen-Langwort enthält dann die Checksumme.

Die Berechnung entspricht eigentlich der, der übrigen Verwaltungsblöcke, nur können wir uns hier das Löschen der alten Checksumme sparen, da sie im ersten Langwort steht. Wir beginnen die Subtraktion einfach erst beim zweiten Langwort. Die Berechnung der Bitmap-Checksumme sieht in Assembler so aus:

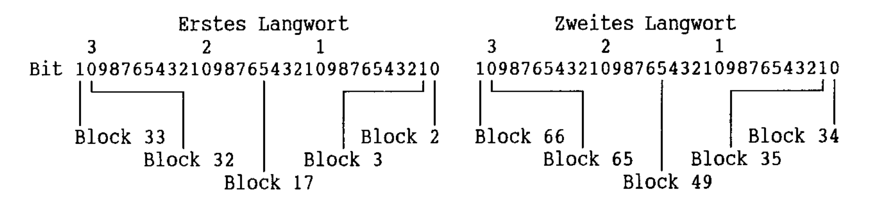
```
        clr.l   d0          ; Lösche Summen-Langwort
        lea     4(a0),a1    ; Beginne Subtraktion bei Blockbeginn+4
        moveq   #126,d1     ; Bearbeite 127 Langwörter
m1:     sub.l   (a1)+,d0    ; Nächstes Langwort subtrahieren
        dbra    d1,m1       ; Schleife
        move.l  d0,(a0)     ; Berechnete Checksumme eintragen
```

Bild 12.2: Berechnung der Bitmap-Checksumme

Aufbau der Bitmap-Daten

Nun zum Aufbau der eigentlichen Bitmap-Daten. Sie beginnen im zweiten Langwort des Bitmap-Blocks. Von hier ab wird jedem Diskblock ein Bit zugeordnet. Wichtig: Die ersten beiden Blöcke der Diskette (Bootblock) tauchen in der Bitmap nicht auf, da sie für das Filesystem quasi gesperrt sind! Der erste Block, der in der Bitmap verzeichnet ist, ist Nr. 2 (Zählung beginnt bei 0).

Jedes Langwort des Bitmap-Blocks kann man nun als Gruppe ansehen. Innerhalb dieser Gruppe wird dem niederwertigsten Bit (Nr. 0) der erste Diskblock der Gruppe zugeordnet, dem höchstwertigsten Bit (Nr. 31) der letzte Block der Gruppe. Im nächsten Langwort wird diese Folge dann fortgesetzt. Die Zuordnung in den ersten beiden Bitmap-Langworten nach der Checksumme sieht also so aus:



Die weiteren Langworte sind analog aufgebaut. Ein gesetztes Bit bedeutet, daß der entsprechende Block frei ist, bei einem gelöschten Bit ist er belegt. Die Anzahl der Diskblöcke, die sich in einem Bitmap-Block verwalten lassen, berechnet sich zu: 127 (Langworte) * 32 (Bits pro Langwort) = 4064 Blöcke. Das entspricht etwa 2 MB Daten.

Da eine Floppy-Disk 1760 Blöcke umfaßt, reicht ein Bitmap-Block vollkommen aus, er wird sogar noch nicht einmal zur Hälfte benötigt. Bei anderen Speichermedien, z.B. Festplatten, ist das aber anders. Deshalb enthält der Rootblock 26 Einträge für Bitmap-Blöcke. Das erhöht die maximale Blockzahl auf 4064 * 26 = 105664 Blöcke, das sind ca. 52 MB. Bei der Benutzung des alten Filesystems ist die maximale Größe eines Speichermediums also 52 MB, mehr kann die Bitmap nicht verwalten. Da diese Einschränkung recht ärgerlich ist, bietet das FFS, das ja hauptsächlich für Festplatten eingesetzt wird, hier eine Verbesserung, auf die wir im nächsten Abschnitt eingehen werden.

Folgende Befehlsfolge berechnet (für eine Floppy-Disk) die Nummer des Bytes und des Bits in der Bitmap, unter dem ein bestimmter Block (dessen Nummer in d0 erwartet wird) zu finden ist. Die Bytenummer, die als Offset zum Bitmap-Block-Anfang zu sehen ist, steht dann in d1 und die Bitnummer in d2.

```
sub.l    #2,d0        ; Bootblock steht nicht in Bitmap
divu     #32,d0       ; Ergebnis = Langwort, Rest = Bit
clr.l    d1           ; Zielregister löschen
clr.l    d2
move.w   d0,d1        ; Langwort nach d1
add.w    #1,d1        ; Checksumme überspringen
asl      #2,d1        ; Aus Langwort mach' Byte
swap     d0           ; Oberes Wort (Rest) nach unten
move.w   d0,d2        ; Bit nach d2
```

Bild 12.3: Berechnung der Byte- und Bitnummer eines Blocks

Dazu ist noch folgendes zu sagen. Da bei der Angabe einer Blocknummer die beiden Bootblöcke mit einbezogen werden, diese in der Bitmap aber nicht enthalten sind, muß die gewünschte Blocknummer zunächst um 2 verkleinert werden. Nach dem DIVU-32-Befehl steht das Ergebnis, also die Langwortnummer, im unteren Wort von d0 und der Rest, die Bitnummer, im oberen. Die Langwortnummer wird in den Byte-Offset umgerechnet, indem das Checksummen-Langwort am Blockanfang übersprungen und dann die Langwortnummer mit 4 malgenommen wird. Die Bitnummer kann direkt aus dem Rest-Wort in d0 übernommen werden.

Um nun zu prüfen, ob ein bestimmter Diskblock frei ist, reicht nach der Durchführung obiger Berechnung ein MOVE- und ein BTST-Befehl (in a0 stehe der Blockanfang der Bitmap):

```
move.l   0(a0,d1),d0    ; Blockanfang plus Byte-Offset
btst     d2,d0          ; Prüft Bit d2 im Langwort d0
bne      blockfrei      ; Bit gelöscht -> Block frei
beq      blockbelegt    ; Bit gesetzt -> Block belegt
```

Bild 12.4: Test, ob ein Diskblock belegt ist

Das BM-Flag (Bitmap Valid) im Rootblock gibt an, ob die Bitmap, so wie sie in den Blöcken steht, korrekt ist oder nicht. Vor jedem Schreibzugriff auf die Diskette, der eventuell dazu führt, daß Blöcke belegt oder freigegeben werden, wird das BM-Flag gelöscht und erst nach Beendigung des Schreibens, unmittelbar nach dem Aktualisieren der Bitmap, wieder gesetzt. Falls nun der Schreibvorgang durch eine äußere Störung (Absturz, Reset etc.) unterbrochen wird, ist die Bitmap als ungültig gekennzeichnet. Bevor erneut auf die Diskette geschrieben werden kann, muß sie erst wieder gültig gemacht, d.h. neu berechnet werden, da das Filesystem ansonsten nicht weiß, welche Blöcke es denn nun belegen darf.

Diese Neuberechnung geht so vor sich: Zuerst werden alle Blöcke als frei gekennzeichnet. Dann geht das Filesystem, ausgehend vom Rootblock, sämtliche Verzeichnisse und eventuellen Unterverzeichnisse und darin enthaltenen Dateien durch und kennzeichnet alle auf diese Weise gefundenen, zu einer korrekten Struktur gehörigen Blöcke, als belegt. Diese neue Bitmap wird in die im Rootblock eingetragenen Bitmap-Blöcke geschrieben, und die Bitmap ist restauriert. Dieser Vorgang wird unmittelbar beim Einlegen einer Diskette mit ungültiger Bitmap eingeleitet und dauert bis zu einer halben Minute. Falls das Filesystem während der Neuberechnung auf einen Fehler in der Diskstruktur (falscher Blocktyp, Block doppelt benutzt, Checksummenfehler etc.) stößt, wird der allseits geliebte Requester "Error validating Disk ... Use Diskdoctor to correct it" ausgegeben.

Der BitMapList-Block (nur beim FFS)

Nun zu der Verbesserung im FFS bezüglich der 52 MB-Einschränkung: Wenn die 26 Bitmap-Blöcke nicht ausreichen, wird in den 26. Bitmapblock-Zeiger im Rootblock statt eines Bitmap-Blocks ein Zeiger auf einen BitMapList-Block eingetragen, der folgendermaßen aussieht:

Der Aufbau des BitMapList-Blocks

Offsets Long	Byte	Eintrag	Bedeutung
1	4	BitMapBlock	Zeiger auf Bitmap-Blöcke
2	8	...	
...	...	...	
126	504		
127	508	NextBMList	Zeiger auf nächsten BML-Block

Dieser Block besteht also aus 127 Zeigern auf Bitmap-Blöcke (was eine Kapazität von 516128 Blöcken oder 252 MB ausmacht), und falls das immer noch nicht ausreicht, zeigt das 128. Langwort auf den nächsten BitMapList-Block (wieder 252 MB). Die Kapazität des Datenträgers ist also nun, jedenfalls aus Sicht der Bitmap-Verwaltung, unbeschränkt.

Dieser Block hat, wie aus dem Diagramm ersichtlich ist, keine Checksumme. Der BitMapList-Block ist neben dem FFS-Datenblock (kommt noch) der einzige Blocktyp, für den dies gilt.

12.1.4 Der Directory-Block

Jedes Verzeichnis auf einer Diskette wird mit Hilfe eines solchen Blocks verwaltet. Schauen wir uns seinen Aufbau an:

Der Aufbau des Directory-Blocks

Offsets Long	Byte	Eintrag	Bedeutung
0	0	2	Primärer Blocktyp: T.SHORT
1	4	Header Key	Zeiger auf den Block selbst
2	8	0	---
3	12	0	---
4	16	0	---
5	20	Checksum	Checksumme des Blocks

6	24	Hash Table	Hash-Tabelle (Zeiger auf Blöcke der Objekte im Verzeichnis)
..	...	...	
77	308		
78	312	0	---
79	316	0	---
80	320	Protect	Protection Bits (Schutzstatus)
81	324	0	---
82	328	Comment	Verzeichnis-Kommentar (BCPL-String)
...	...	...	
104	416		
105	420	Days	Datum und Uhrzeit des letzten Schreibzugriffs (im DateStamp-Format)
106	424	Mins	
107	428	Ticks	
108	432	Directory Name	Verzeichnisname (BCPL-String)
...	...	...	
120	480		
121	484	0	---
122	488	0	---
123	492	0	---
124	496	Hashchain	Nächster Hash-Block (siehe unten)
125	500	Parent	Zeiger auf übergeordnetes Verzeichn.
126	504	0	---
127	508	2	Sekundärer Blocktyp: ST.USERDIR

Blocktyp Die Kennwerte für den Blocktyp Directory-Block sind T.SHORT und ST.USERDIR (beides 2).

Header Key Dieser Zeiger, der auf den Block selbst weist, dient als Kontrolle, ob der Block korrekt ist oder durch irgendwelche User-Umkopier-Aktionen von einer anderen Stelle hierher geraten ist.

Checksum Die Berechnung der Directory-Block-Checksumme läuft exakt genauso ab wie beim Rootblock.

Hash Table Auch ein Directory-Block hat eine Hash-Tabelle. Auf das Verfahren des Hashings gehen wir nach dieser Beschreibung ein.

Protect Die Bits in diesem Langwort entsprechen denen, die man beim DOS-SetProtection-Aufruf (siehe DOS-Kapitel) verwendet.

Comment An die Stelle der Bitmap-Blockzeiger des Rootblocks tritt im Directory-Block der Comment-BCPL-String.

Days/Mins/Ticks Der Zeitpunkt des letzten Schreibzugriffs wird auch hier im DateStamp-Format festgehalten.

Directory Name Der Name als BCPL-String.

Hashchain Dieser Eintrag spielt im Zusammenhang mit dem Hashing eine Rolle.

Parent Bis auf den Rootblock enthält jeder Block einen Zeiger auf den übergeordneten Block, beim Directory-Block ist das der Verwaltungsblock des übergeordneten Verzeichnisses.

Berechnung der Blocknummer aus dem Namen

Das Amiga-Filesystem ist derzeit so ziemlich das schnellste Filesystem, wenn es darum geht, eine Datei oder ein Verzeichnis mit bekanntem Namen auf einem Datenträger zu lokalisieren. Es verwendet die Methode des "Hashings" (die Nummer des Verwaltungsblocks wird aus dem Namen berechnet).

In jedem Directory-Block (auch im Rootblock) gibt es eine Tabelle, die aus 72 Langwörtern besteht. Sie nennt sich die Hash-Tabelle (ihre Größe ist auch im Eintrag "HT Size" des Rootblocks abgelegt). In diese 72 Langwörter werden nun Zeiger auf die Verwaltungsblöcke der zum Verzeichnis gehörigen Dateien oder Unterverzeichnisse eingetragen, und zwar nicht beliebig, alphabetisch oder in Reihenfolge des Anlegens, sondern nach einer ganz bestimmten Formel.

Die Platznummer eines Blockzeigers in der Hash-Tabelle wird aus dem Objektsnamen errechnet, und zwar auf folgende Weise:

1. Den Startwert der Hashnummer bildet die Länge des Namens. Zu ihm werden alle weiteren Werte hinzuaddiert.
2. Alle Kleinbuchstaben (a-z, nicht aber Umlaute und ß) werden in Großbuchstaben umgewandelt.
3. Der bisherige Hash-Wert wird mit 13 multipliziert.
4. Der ASCII-Code des (eventuell umgewandelten) nächsten Zeichens wird zum Hashwert hinzuaddiert.
5. Nach der Addition wird der Hashwert mit $7FF AND-verknüpft.
6. Wiederholung der Punkte 3 - 5 für alle Zeichen im Namen.
7. Der Hashwert wird durch die Größe der Hash-Tabelle (72) Modulo-dividiert (d.h. der Rest der Ganzzahl-Division wird als Ergebnis verwendet).

Die solchermaßen berechnete Nummer gibt den Platz des Hashtabellen-Eintrags (0-71) für diesen Namen an. Um sie in einen Byte-Offset umzurechnen, muß 6 (das Start-Langwort der Hash-Tabelle) zur Nummer hinzuaddiert und die Summe mit 4 multipliziert werden. Das Berechnungsverfahren gilt natürlich nur für reine Datei- oder Verzeichnisnamen, ohne irgendwelche Pfadangaben.

Da die Hash-Tabelle 72 Einträge umfaßt, in der Praxis aber sicherlich mehr als 72 verschiedene Dateinamen auftreten können, sind Doppelbelegungen unvermeidlich. In diesem Fall wird, falls ein Blockzeiger in der Hash-Tabelle schon belegt ist, der Blockzeiger auf das neue Objekt im Eintrag

"Hashchain", zu Deutsch Hash-Verkettung, abgelegt. Sucht man nun nach einer bestimmten Datei und hat man, gemäß ihrer Hashtabellen-Platznummer, den dort gefundenen Block eingelesen, muß man sich vergewissern, ob es sich bei dem gelesenen Block wirklich um den gewünschten handelt. Wenn nicht, muß der Block der nächsten Datei, deren Namen auch auf den berechneten Hash-Wert paßt, aus dem Hashchain-Eintrag ausgelesen und geladen werden.

12.1.5 Der Fileheader-Block

Nun kommen wir langsam zum interessanten Teil: den Dateien. Jede Datei wird über einen Fileheader-Block verwaltet, dessen Aufbau wir uns jetzt ansehen wollen:

Der Aufbau des Fileheader-Blocks

Offsets Long	Byte	Eintrag	Bedeutung
0	0	2	Primärer Blocktyp: T.SHORT
1	4	Header Key	Zeiger auf den Block selbst
2	8	Highest Seq	Anzahl verwendeter Block-Zeiger
3	12	0	---
4	16	First Data	Zeiger auf ersten Datenblock
5	20	Checksum	Checksumme des Blocks
6 7 .. 76 77	24 28 ... 304 308	Data Blk 71 Data Blk 70 ... Data Blk 1 Data Blk 0	Zeiger auf die ersten 72 Datenblöcke (die weiteren werden in FileList-Blöcken abgelegt)
78 79	312 316	0 0	--- ---
80	320	Protect	Protection Bits (Schutzstatus)
81	324	Byte Size	Größe der Datei in Bytes
82 ... 104	328 ... 416	Comment ...	Datei-Kommentar (BCPL-String)
105 106 107	420 424 428	Days Mins Ticks	Datum und Uhrzeit des letzten Schreibzugriffs (im DateStamp-Format)

108	432	File Name	Dateiname (BCPL-String)
...	...	...	
120	480		
121	484	0	---
122	488	0	---
123	492	0	---
124	496	Hashchain	Nächster Hash-Block
125	500	Parent	Zeiger auf übergeordnetes Verzeichn.
126	504	Extension	Zeiger auf ersten FileList-Block
127	508	-3	Sekundärer Blocktyp: ST.FILE

Blocktyp Als Kennzeichnung für einen Fileheader-Block werden die Werte 2 bzw. -3 für T.SHORT bzw. ST.FILE verwendet.

Header Key Der Zeiger auf den Block selbst dient wieder der Fehlererkennung.

Highest Seq Hier wird verzeichnet, wieviele der maximal 72 Zeiger auf Datenblöcke verwendet werden. Das bezieht sich aber nur auf die Blockzeiger im Fileheader-Block. Wenn eine Datei länger als 72 Blöcke ist und daher ein Filelist-Block verwendet wird, kommt in den Highest-Seq-Eintrag des Fileheader-Blocks trotzdem eine 72, da dort ja nur 72 Blockzeiger vorhanden sind, die benutzt werden können.

First Data Getrennt von den Blockzeigern wird hier der Zeiger auf den ersten Datenblock zusätzlich auch eingetragen.

Checksum Zur Berechnung der Fileheader-Block-Checksumme wird das selbe Verfahren verwendet wie beim Rootblock.

Data Blk 0 - 71 Die Datenblöcke stehen quasi in "umgekehrter" Reihenfolge in der Blockzeiger-Tabelle. Der erste Datenblock wird also im letzten Eintrag der Tabelle verzeichnet.

Protect Entspricht dem Protect-Eintrag im Directory-Block.

Byte Size Zum schnellen Ermitteln der Größe einer Datei in Bytes steht selbige hier.

Comment Der Datei-Kommentar als BCPL-String.

Days/Mins/Ticks Zeitpunkt des letzten Schreibzugriffes (der letzten Veränderung) auf die Datei im DateStamp-Format.

File Name Der Dateiname als BCPL-String.

Hash Chain Da es in einem Verzeichnis auch mehrere Dateinamen mit dem gleichen Hash-Wert geben kann, sind auch die Fileheader-Blöcke mit einem Zeiger auf die nächste Datei (bzw. das nächste

	Verzeichnis) mit zum Hash-Wert passendenden Namen ausgestattet.
Parent	Hier wird ein Zeiger auf den Verwaltungsblock des Verzeichnisses eingetragen, in dem sich die Datei befindet.
Extension	Sollte die Datei mehr als 72 Blöcke groß sein, werden die weiteren Blöcke in einem FileList-Block verwaltet (die Blockzeiger-Tabelle des Fileheader-Blocks umfaßt maximal 72 Einträge). An diese Stelle wird der Zeiger auf den ersten FileList-Block eingetragen.

Der Aufbau dieses Blocktyps dürfte damit klar sein.

12.1.6 Der Filelist-Block

Im Fileheader-Block ist nur Platz für 72 Datenblock-Zeiger, die Datei könnte demnach nur 488 * 72 = 35136 Bytes groß sein (warum 488 und nicht 512, werden wir gleich noch sehen). Um diesen Umstand zu umgehen, wird im Falle einer Datei, die mehr als 72 Blöcke umfaßt, ein Filelist-Block angelegt. Er hat fast den selben Aufbau wie der Fileheader-Block, einige Einträge sind jedoch unbelegt. Hier das Aufbau-Diagramm:

Der Aufbau des Filelist-Blocks

Offsets Long	Byte	Eintrag	Bedeutung
0	0	16	Primärer Blocktyp: T.LIST
1	4	Header Key	Zeiger auf den Block selbst
2	8	Block Count	Anzahl der verwendeten Blockzeiger
3	12	0	---
4	16	0	---
5	20	Checksum	Checksumme des Blocks
6	24	Data Blk N+71	Zeiger auf weitere 72 Datenblöcke
7	28	Data Blk N+70	
..	...	...	
76	304	Data Blk N+1	
77	308	Data Blk N+0	
78	312	0	---
79	316	...	
...	...		
124	496		
125	500	Parent	Zeiger auf Fileheader-Block

126	504	Extension	Zeiger auf nächsten FileList-Block
127	508	-3	Sekundärer Blocktyp: ST.FILE

Blocktyp Als Kennzeichnung für einen Filelist-Block werden die Werte 16 bzw. -3 für T.LIST bzw. ST.FILE verwendet.

Header Key Der Zeiger auf den Block selbst dient wieder der Fehlererkennung.

Block Count Hier wird verzeichnet, wieviele der maximal 72 Zeiger auf Datenblöcke verwendet werden.

Checksum Zur Berechnung der Filelist-Block-Checksumme wird das selbe Verfahren verwendet wie beim Rootblock.

Data Blk N+0 - N+71 Die Datenblöcke stehen auch hier quasi in "umgekehrter" Reihenfolge in der Blockzeiger-Tabelle.

Parent Hier wird ein Zeiger auf den Fileheader-Block eingetragen, zu dem diese Filelist gehört.

Extension Sollten die bisherigen Datenblöcke noch nicht ausreichen, wird hier der Zeiger auf den nächsten FileList-Block eingetragen.

Damit wäre der Filelist-Block klar.

12.1.7 Der Data-Block

Nun sind wir nach den ganzen Verwaltungsblöcken endlich dort angekommen, wo die Daten wirklich zu finden sind. Neben den reinen Daten werden in einen Data-Block aber auch noch ein paar zusätzliche Informationen gespeichert. 6 Langworte von den 128 eines Blockes werden dafür aufgewendet. Der Datenmenge, die in einem Block Platz hat, mag das etwas abträglich sein, aber die Zusatzdaten sind äußerst nützlich, wenn es darum geht, teilweise defekte Diskstrukturen wieder zu restaurieren.

Dank der Verwaltungsinformationen in den Datenblöcken können nämlich sämtliche sonstigen Directory-, Fileheader- und Filelist-Blöcke gelöscht sein, die Dateien können aber trotzdem noch gerettet werden (bis auf den Namen, das Schreibdatum, den Schutzstatus und die Zuordnung zu den Verzeichnissen).

Der Aufbau des Data-Blocks

Offsets Long	Byte	Eintrag	Bedeutung
0	0	8	Primärer Blocktyp: T.DATA
1	4	Header Key	Zeiger auf Fileheader-Block
2	8	Seq Num	Position des Blocks in der Datei
3	12	Data Size	Zahl der benutzten Bytes im Block
4	16	Next Data	Zeiger auf nächsten Datenblock
5	20	Checksum	Checksumme des Blocks
6	24	Data	488 Bytes Daten
7	28	...	
..	...	...	
127	508		

Blocktyp Der Data-Block hat nur ein Blocktyp-Langwort, nämlich das am Blockanfang. Hier ist eine 8 für T.DATA zu finden.

Header Key Beim Data-Block steht hier nicht ein Zeiger auf den Block selbst, sondern auf den Fileheader-Block der Datei, zu der der Data-Block gehört.

Seq Num Die Position des Datenblocks in der Blockkette der Datei.

Data Size Gibt an, wieviele der 488 Bytes des Blocks verwendet werden (eine Datei muß ja nicht genau auf Blockgrenze enden).

Next Data Zeiger auf den nächsten Datenblock.

Checksum Die Berechnung erfolgt wie beim Rootblock.

Data Endlich haben wir die Daten erreicht ...

Der Data-Block beim FFS

Neben dem Verwaltung der Bitmap-Blöcke unterscheidet sich das FFS auch im Aufbau der Datenblöcke vom OFS. Beim FFS werden im Data-Block keine Verwaltungsinformationen gespeichert. Die vollen 512 Bytes werden zur Datenspeicherung verwendet.

Diese Tatsache hat sowohl Vor- als auch Nachteile. Die Vorteile sind zum einen die größere Speicherkapazität (24 Bytes mehr pro Block, das ergibt pro Megabyte Speicherplatz 48 KB mehr), zum anderen eine höhere Ladegeschwindigkeit. Beim alten Filesystem müssen nämlich die Daten jedes Datenblocks nach dem Laden noch umkopiert werden, um die Verwaltungsinformationen auszufiltern. Das ist beim FFS nicht nötig, hier können die Daten "am Stück" in den gewünschten Speicherbereich gelesen werden.

Nachteile gibt es eigentlich nur einen: die Datensicherheit. Wenn beim FFS einmal die Verwaltungsblöcke (Fileheader oder Filelist) zerstört sind, gibt es so gut wie keine Möglichkeit mehr, die Datei noch zu restaurieren, obwohl die Daten selbst womöglich noch wohlbehalten auf der Disk stehen. Man weiß aufgrund der zerstörten Blockliste einfach nicht mehr, welche Diskblöcke zur Datei gehörten. Beim alten Filesystem wäre das kein Problem, da in jedem Datenblock angegeben ist, zu welcher Datei er gehört (Eintrag "Header Key") und der wievielte Block der Datei er ist (Eintrag "Seq Num"). Beim FFS gibt es diese Einträge nicht, deshalb sieht es bei einer zerstörten Blockliste ziemlich übel aus.

Bis einschließlich zur Betriebssystemsversion 1.3 wird das FFS nur für Festplatten benutzt. Ab Kickstart 2.0 ist es auch möglich, Disketten unter FFS zu beschreiben.

12.1.8 Programmierung des Bootblocks

Zu Beginn dieses Kapitels haben wir den Bootblock im Zusammenhang mit der Filesystem-Kennung schon einmal kurz angesprochen. Nun wollen wir uns den Aufbau dieser beiden Blöcke, die eigentlich nicht zum Filesystem gehören, anschauen. Zuerst das Diagramm:

Der Aufbau des Bootblocks

Offsets Long	Byte	Eintrag	Bedeutung
0	0	ID	Filesystem-Kennung
1	4	Checksum	Bootblock-Checksumme
2	8	Dosblock	Zeiger auf Rootblock (880)
3	12	Program	1012 Bytes Bootprogramm
4	16	...	
..	...	...	
255	1020		

ID	Entweder $444F5300 für eine OFS-Diskette oder $444F5301 für eine FFS-Diskette.
Checksum	Die Berechnung der Boot-Checksumme läuft anders ab als die bisher bekannten Checksummen. Wir werden gleich darauf kommen.
Dosblock	Hier wird ein Zeiger auf den Rootblock (bei Disketten Block 880) erwartet.
Program	Für das Bootprogramm müssen bestimmte Regeln erfüllt sein. Wir werden gleich darauf zu sprechen kommen.

Damit eine Diskette vom Filesystem anerkannt wird, muß nur der Eintrag "ID" gesetzt sein. Zur Ausführung eines Systemstarts von der Diskette aber müssen auch die übrigen Daten vorhanden sein. Die Checksumme spielt in diesem Zusammenhang eine wichtige Rolle: Nur wenn sie korrekt ist, kann der Systemstart erfolgen, ansonsten wird die Diskette behandelt, als wäre nur die ID-Kennung und kein Bootprogramm darauf.

Die Berechnung der Boot-Checksumme

Da der Bootblock aus zwei Disk-Blöcken besteht, muß die Checksumme auch über beide Blöcke berechnet werden. Das Verfahren sieht folgendermaßen aus:

1. Lösche das Summen-Langwort und den eventuell vorhandenen bisherigen Checksummen-Eintrag.
2. Addiere ein Langwort des Blocks zum Summen-Langwort.
3. Wenn es bei der Addition einen Überlauf gegeben hat (Abfrage mit dem BCC- oder BCS-Befehl), addiere zusätzlich eine 1 zum Summen-Langwort.
4. Wiederholung der Punkte 2 und 3 für alle Langwörter des Bootblocks.
5. Die Checksumme wird durch NOT-Verknüpfung des Summen-Langworts ermittelt.

Die berechnete Checksumme muß wieder ins entsprechende Block-Langwort eingetragen werden. Die Berechnung in Assembler könnte so aussehen (Zeiger auf Bootblock in a0):

```
        clr.l    d0           ; Lösche Summen-Langwort
        clr.l    4(a0)        ; Lösche alte Checksumme
        move.l   a0,a1        ; Blockzeiger zur Bearbeitung nach a1
        move.w   #255,d1      ; Bearbeite 256 Langwörter (2 Blöcke)
m1:     add.l    (a1)+,d0     ; Nächstes Langwort subtrahieren
        bcc      m2           ; Wenn kein Überlauf
        addq     #1,d0        ; Sonst noch eine 1 aufaddieren
m2:     dbra     d1,m1        ; Schleife
        not.l    d0           ; d0 NOT-verknüpfen
        move.l   d0,4(a0)     ; Berechnete Checksumme eintragen
```

Bild 12.5: Berechnung der Bootblock-Checksumme

Anforderungen an das Boot-Programm

Wenn man ein eigenes Bootprogramm schreiben will, muß man zunächst dafür sorgen, daß die drei Verwaltungs-Langwörter vor dem eigentlichen Programm stehen. In Assembler sieht das so aus:

```
        dc.l     $444f5300    ; DOS-Kennung
        dc.l     0            ; Für spätere Checksumme
        dc.l     880          ; DOS-Rootblock
```

```
begin:  ...                  ; Hier beginnt das Programm
```

Das eigentliche Programm darf keine, sich auf das Programm beziehenden, absoluten Adressierungen verwenden. Eine Befehlsfolge wie

```
        move.l   gfxbase,a6      ; Im Bootprogramm FALSCH!
```

ist also nicht erlaubt. Statt dessen müßte man schreiben:

```
        lea      gfxbase(pc),a0 ; Lade Adresse PC-relativ
        move.l   (a0),a6        ; Inhalt der Adresse nach a6
```

Alle Adressierungen, die sich auf das Programm selbst beziehen, müssen also PC-relativ erfolgen. Das liegt daran, daß man nicht weiß, an welche Adresse das Bootprogramm vom System geladen wird. Das Programm muß also lageunabhängig sein. Gewöhnlich wird dies durch die schon erwähnte Tabelle mit allen verwendeten absoluten Adressen, die an das Programm (vom Assembler) angehangen wird, erledigt. Für das Bootprogramm gibt es aber eine solche Tabelle nicht. Wir müssen also selbst für die Lageunabhängigkeit sorgen, und das tun wir eben durch die PC-relative Adressierung.

Des weiteren gibt es gewisse Einschränkungen was die Benutzbarkeit der Libraries angeht. Eine Benutzung der DOS-Library ist im Bootprogramm verboten, da selbige noch nicht initialisiert ist. Auch müssen Sie auf alle Intuition-Routinen, die mit Screens, Windows und Dazugehörigem arbeiten, verzichten. Die Routine DisplayAlert können Sie allerdings einsetzen, wenn Sie dies wünschen. Möglich ist die Benutzung der Exec- und Graphics-Library, was für die meisten Zwecke wohl ausreichen dürfte.

Eine weitere Einschränkung ergibt sich aus der Größe des Bootblocks: Da er nur zwei Diskblöcke umfaßt, von denen drei Langworte für die Verwaltung entfallen, darf ein Bootprogramm höchstens 1012 Bytes lang sein, es sei denn, Sie laden andere Diskblöcke vom Bootblock aus nach (natürlich "direkt" über das Trackdisk-Device, nicht über DOS!).

Das normale DOS-Bootprogramm

Wie schon erwähnt, übernimmt das Standard-DOS-Bootprogramm Initialisierungsaufgaben. Es ermittelt die Startadresse der Resident-Struktur der DOS-Library und gibt diese in a0 zurück. Dies wird von der aufrufenden Systemroutine als Zeichen angesehen, daß das DOS einsatzbereit ist, und außerdem wird die Resident-Adresse zur Weiterverarbeitung benötigt. Was wir Ihnen sagen wollen, ist, daß diese Schritte zum Start des Systems unerläßlich sind. Ein Bootprogramm, daß "nur" das System starten soll, darf also nicht einfach "RTS" heißen, sondern muß die oben genannten Schritte durchführen. Damit Sie wissen, wie das ganze in Assembler aussieht, hier nun das Standard-DOS-Bootprogramm als Listing:

```
        dc.l    $444f5300       ; Filesystem-Kennung
        dc.l    $c0200f19       ; Checksumme
        dc.l    880             ; DOS-Rootblock

        lea     dosname(pc),a1  ; Name der DOS-Lib
        jsr     -96(a6)         ; Exec-Routine FindResident
        tst.l   d0              ; Resident gefunden?
        beq     m2              ; Wenn nein
        move.l  d0,a0           ; Resident-Start nach a0
        move.l  22(a0),a0       ; Zeiger auf Init-Code nach a0
        moveq   #0,d0           ; d0 löschen als OK-Zeichen
m1:     rts

m2:     moveq   #255,d0         ; 255 nach d0 als Fehler-Zeichen
        bra     m1

dosname:      dc.b    "dos.library",0
              dcb.b   974,0          ; Füll-Null-Bytes
```

Bild 12.6: Der Standard-DOS-Bootblock

Die 974 Füll-Nullbytes sind nötig, damit keine zufälligen Werte hinter dem Programm folgen. Grundsätzlich wäre es zwar egal, wenn nach dem Programm zufällige Bytes im Bootblock stünden, aber die hier angegebene Checksumme würde dann nicht stimmen.

Falls Sie also ein Bootprogramm schreiben möchten, das nach seiner Beendigung das System wirklich booten läßt, muß die obige Befehlsfolge irgendwo im Programm untergebracht werden (am besten unmittelbar vor dem Ende, damit die Register nicht wieder überschrieben werden). Das System erwartet vom Bootprogramm in a0 den Init-Code-Zeiger und in d0 eine 0 als OK-Kennzeichen.

Es wird Sie bestimmt wundern, daß in der zweiten Programmzeile die FindResident-Routine einfach so, ohne voriges "move.l 4,a6" aufgerufen werden kann. Das ist möglich, da das System die Register vor dem Aufruf des Bootprogramms teilweise vorbelegt hat. In a6 steht auf diese Weise der Zeiger auf die Exec-Basis und in a1 der Zeiger auf eine vollständig initialisierte IO-Request-Struktur für das Laufwerk DF0:. Das ist im Grunde eine feine Sache, denn so können wir ohne große Vorarbeit über das Trackdisk-Device von DF0: lesen.

Als nächstes nun ein Bootblock-Rahmenprogramm, das Sie für Ihre Bootprogramme verwenden können. Es enthält alle nötigen Vorkehrungen, damit das System nach seiner Beendigung das DOS bootet.

```
        dc.l    $444f5300       ; Filesystem-Kennung
        dc.l    0               ; für spätere Checksumme
```

```
        dc.l     880              ; DOS-Rootblock
        movem.l  d0-d7/a0-a6,-(sp)     ; Alle Register sichern
        bsr      main             ; Hauptroutine anspringen
        movem.l  (sp)+,d0-d7/a0-a6     ; Register zurückholen

        lea      dosname(pc),a1 ; DOS-Resident holen (die Fehlerabfrage
        jsr      -96(a6)          ; können wir uns sparen)
        move.l   d0,a0            ; Resident-Start nach a0
        move.l   22(a0),a0        ; Zeiger auf Init-Code nach a0
        moveq    #0,d0            ; d0 löschen als OK-Zeichen
m1:     rts

dosname:      dc.b    "dos.library",0

main:   ...                       ; Hier steht die Hauptroutine
```

Bild 12.7: Ein Boot-Rahmenprogramm

Vor dem Anspringen der Hauptroutine retten wir alle Register. Das ist zwar nicht unbedingt nötig, aber sicherer. Nach der Hauptroutine werden die Befehle des Standard-DOS-Bootblocks ausgeführt, wobei wir uns die Abfrage, ob der DOS-Resident vorhanden ist, schenken können (er ist immer vorhanden, es sei denn, Ihr ROM ist beschädigt).

Als Abschluß des Bootblock-Abschnitts stellen wir nun ein Beispielprogramm vor. Es handelt sich dabei um eine abgewandelte Version des View-Demoprogramms aus dem Graphics-Kapitel (Zeichnen eines Ellipsen-Musters auf einem eigenen View). Die View-Einrichtung ist notwendig, da wir im Bootblock noch keine Intuition-Screens verwenden können.

Das Programm erfüllt zwei Funktionen: Es enthält das eigentliche Bootprogramm und einen Programmteil, der das Bootprogramm per Trackdisk-Device-Zugriff auf die Diskette bringt (wobei vorher die Checksumme berechnet wird). Das komplette Programm finden Sie auf der Diskette unter "PRG_12_1.S", wir drucken hier die Hauptroutine nicht ab.

```
* Programm 12.1 (Auszug): Demonstration eines Bootprogrammes

* EQUs für Installationsprogramm

ExecBase      =       4
FindTask      =       -294
OpenDevice    =       -444
DoIO          =       -456
CloseDevice   =       -450

* EQUs für Bootprogramm

FindResident  =       -96
```

```
        ...

* Beginn des Installationsprogramms

        move.l  ExecBase,a6

        sub.l   a1,a1           ; entspricht 'move.l #0,a1'
        jsr     FindTask(a6)    ; Task-Adresse ermitteln
        move.l  d0,port+16      ; In Reply-Port eintragen

        lea     stdio,a1        ; Zeiger auf StdIOReq-Struktur
        move.l  #port,14(a1)    ; Port-Zeiger in StdIO eintragen
        move.l  #0,d0           ; Unit 0
        clr.l   d1              ; Keine Flags
        lea     tddname,a0      ; Zeiger auf TDD-Name
        jsr     OpenDevice(a6)  ; Trackdisk-Device öffnen

        lea     bootprg,a0      ; Start des Bootprg nach a0

        clr.l   d0              ; Checksumme berechnen
        move.l  a0,a1
        move.w  #255,d1
m1:     add.l   (a1)+,d0
        bcc     m2
        addq    #1,d0
m2:     dbra    d1,m1
        not.l   d0
        move.l  d0,4(a0)

        lea     stdio,a1        ; StdIO nach a1
        move.w  #3,28(a1)       ; Kommando: Write
        move.l  a0,40(a1)       ; Adresse: Start Bootprg
        clr.l   44(a1)          ; Startblock: 0
        move.l  #1024,36(a1)    ; Länge: 2 Blocks (1024 Bytes)
        jsr     DoIO(a6)        ; Kommando ausführen

        move.w  #4,28(a1)       ; Kommando: Update
        jsr     DoIO(a6)        ; ausführen

        move.w  #9,28(a1)       ; Kommando: Motor
        clr.l   36(a1)          ; Ausschalten
        jsr     DoIO(a6)        ; Kommando ausführen

        jsr     CloseDevice(a6)         ; TDD schließen

        rts

* Daten für Installation

stdio:          dcb.b   66,0
port:           dcb.b   34,0
tddname:        dc.b    "trackdisk.device",0
                even

                section "",code_c       ; Lade Programm ins Chip-RAM
```

```
                                        ; (weil das TDD nur auf Chip-RAM
                                        ; zugreifen kann)

* Das Boot-Programm, entspricht DrawEllipse-Demo, jetzt aber
* PC-relativ programmiert

bootprg:
        dc.l     $444f5300        ; Filesystem-Kennung
        dc.l     0                ; für spätere Checksumme
        dc.l     880              ; DOS-Rootblock

        movem.l  d0-d7/a0-a6,-(sp)      ; Alle Register sichern
        bsr      main             ; Hauptroutine anspringen
        movem.l  (sp)+,d0-d7/a0-a6      ; Register zurückholen

        lea      dosname(pc),a1 ; DOS-Resident holen (die Fehlerabfrage
        jsr      FindResident(a6)       ; können wir uns sparen)
        move.l   d0,a0            ; Resident-Start nach a0
        move.l   22(a0),a0        ; Zeiger auf Init-Code nach a0
        moveq    #0,d0            ; d0 löschen als OK-Zeichen
        rts

main:   move.l   4,a6             ; Lib öffnen

        ...

filler:       dcb.b   1024,0          ; Damit der Rest des Bootblocks
                                      ; mit Nullen gefüllt ist
```

Programm 12.1 (Auszug)

Die Benutzung des Trackdisk-Devices ist ja schon aus dem Device-Kapitel bekannt. Am eigentlichen Programm hat sich, bis auf die Tatsache, daß es jetzt PC-relativ programmiert ist, nicht viel geändert. Der Bootblock-Kopf für die Rückkehr zum DOS ist natürlich dazugekommen.Legen Sie vor dem Start des Programms eine (nicht schreibge schützte) Diskette in DF0:. Vergewissern Sie sich aber, daß sie keinen wichtigen Bootblock enthält (der Workbench-Screen muß beim Laden erscheinen). Das Programm schreibt die Ellipsen-Routine in den Bootblock. Wenn Sie das System nun von dieser Diskette starten, erscheint sofort nach dem Einlegen das Ellipsen-Muster am Bildschirm. Ein netter Effekt, nicht wahr?

12.2 Die Basisstruktur der DOS-Library

Wie man von dieser Library wohl erwarten kann, finden sich in den Einträgen der Basisstruktur Informationen, die sich mit Speichermedien, Verzeichnissen usw. beschäftigen. Werfen wir einen Blick auf sie:

12.2.1 Die DOSLibrary-Struktur

```
00    ds.b    dl_lib,34            ; Library-Struktur
34    dc.l   *dl_Root              ; Zeiger auf Root-Node
38    dc.l   *dl_GV                ; Zeiger auf 'Global Vector'
42    dc.l    dl_A2                ; DOS-interne Register-
46    dc.l    dl_A5                ; zwischenspeicher
50    dc.l    dl_A6
54            dl_SIZEOF
```

dl_lib
Die schon bekannte Library-Struktur zur Verwaltung der Library durch Exec.

***dl_Root**
Ein Zeiger auf eine weitere Struktur, die im Anschluß an diese beschrieben wird.

***dl_GV**
Dieser Zeiger ist nur für BCPL-Programme wichtig und soll uns nicht weiter interessieren.

dl_A2, dl_A5, dl_A6
Diese Einträge werden vom DOS zur internen Zwischenspeicherung der gleichnamigen Adreßregister verwendet.

12.2.2 Die RootNode-Struktur

Der einzige interessante Eintrag in der Basisstruktur ist der Zeiger auf die 'RootNode', die wir uns auch sogleich ansehen wollen:

```
00    dc.l   *rn_TaskArray         ; BCPL-Zeiger auf CLI-Task-Tabelle
04    dc.l   *rn_ConsoleSegment    ; BCPL-Zeiger auf Konsolen-Handler
08    ds.b    rn_Time,12           ; Systemzeit im DateStamp-Format
20    dc.l   *rn_RestartSeg        ; BCPL-Zeiger auf Disk-Validator
24    dc.l   *rn_Info              ; BCPL-Zeiger auf DosInfo-Struktur
28    dc.l   *rn_FileHandlerSegment; BCPL-Zeiger auf File-Handler
32            rn_SIZEOF
```

***rn_TaskArray**
Der Zeiger weist auf eine Tabelle bestehend aus Langwörtern. Das erste Langwort gibt die Anzahl der folgenden Einträge an. Ab dem zweiten Langwort folgen die Zeiger auf die Prozeß-Strukturen der einzelnen CLI-Tasks.

***rn_ConsoleSegment**
Dieser Zeiger weist auf das Programmsegment für den CLI-Konsolen-Handler (der für die Tastatureingabe und Bildschirmausgabe zuständig ist).

rn_Time
Die hier im DateStamp-Format abgelegte Systemzeit wird ständig aktualisiert.

***rn_RestartSeg**
Hier wird ein Zeiger auf das Programmsegment für den Disk-Validator, der neu eingelegte Disketten auf gültigkeit überprüft, eingetragen.

***rn_Info**
Ein Zeiger auf eine weitere interessante Struktur, DosInfo, die gleich besprochen wird.

***rn_FileHandlerSegment**
Noch ein Programmsegment-Zeiger, diesmal für den File-Handler, der die Verbindung vom DOS zum Filesystem darstellt.

Beachten Sie, daß sich die hier verwendeten BCPL-Zeiger (siehe Kommentare in der Struktur) von "normalen" Zeigern unterscheiden (siehe Abschnitt 4.2.2).

12.2.3 Die DosInfo-Struktur

Die DOS-Basis ist wirklich ganz schon "verzeigert". Nun sind wir schon bei der dritten Unterstruktur angekommen, und es geht noch weiter. Die DosInfo-Struktur beinhaltet zur Zeit nämlich nur einen interessanten Eintrag:

```
00    dc.l   *di_McName       ; Derzeit nicht benutzt
04    dc.l   *di_DevInfo      ; BCPL-Zeiger auf Device-List
08    dc.l   *di_Devices      ; Derzeit nicht benutzt
12    dc.l   *di_Handlers     ; Derzeit nicht benutzt
16    dc.l   *di_NetHand      ; Derzeit nicht benutzt
20            di_SIZEOF
```

***di_McName, *di_Devices, *di_Handlers, *di_NetHand**
Irgendwann war es wohl einmal vorgesehen, den Amiga mit einem Netzwerk-Betriebssystem auszustatten. Dazu ist es aber nicht gekommen, nur diese vier nicht benutzten Einträge der DosInfo-Struktur zeugen von dem Vorhaben.

***di_DevInfo**
Das Zeigern geht munter weiter. DevInfo weist auf eine verkettete Strukturliste, in der alle Geräte (Devices), Disketten (Volumes) und zugewiesenen Verzeichnisse (assigned Directories) aufgeführt sind.

12.2.4 Die DevList-Struktur

Diese Struktur kann drei verschiedene Formen annehmen, je nachdem, ob ein Gerät, eine Diskette oder ein Verzeichnis mit ihr beschrieben wird. Einige Einträge sind in allen Formen gleich, einige sind unterschiedlich.

Die DevList-Struktur für Devices (DeviceNode)

Stellt die DevList-Struktur ein Gerät dar, hat sie den Namen DeviceNode und folgendes Aussehen:

```
00    dc.l   *dn_Next          ; BCPL-Zeiger auf nächste Struktur
04    dc.l    dn_Type          ; Struktur-Typ (bei Devices 0)
08    dc.l   *dn_Task          ; Zeiger auf Device-Task
12    dc.l   *dn_Lock          ; Für Devices nicht benutzt
16    dc.l   *dn_Handler       ; Handler-Dateiname als BCPL-String
20    dc.l    dn_StackSize     ; Stack-Größe für neue Tasks
24    dc.l    dn_Priority      ; Priorität für neue Tasks
28    dc.l   *dn_Startup       ; BCPL-Zeiger auf FileSysStartupMsg
32    dc.l   *dn_SegList       ; BCPL-Zeiger auf Handler-Segment
36    dc.l   *dn_GlobalVec     ; Zeiger auf globalen Vektor
40    dc.l   *dn_Name          ; Name des Devices als BCPL-String
44            dn_SIZEOF
```

***dn_Next**
Die DeviceNode-Strukturen sind zu einer einfach verketteten Liste verbunden, d.h. jede Struktur bis auf die letzte enthält einen Zeiger auf die nächste.

dn_Type
Gibt den Typ der DevList-Struktur an. Für DeviceNode muß hier eine 0 stehen.

***dn_Task**
Zeiger auf den Task, der das Device bedient.

***dn_Handler, *dn_SegList**
In dn_SegList wird ein Zeiger auf das Programmsegment des Device-Handlers eingetragen. Fehlt dieser, so wird der BCPL-String dn_Handler als Name der Handlerdatei interpretiert und diese eingeladen.

dn_StackSize, dn_Priority
Hier werden die Stackgröße und die Priorität der vom Device aus neu gestarteten Tasks eingetragen.

***dn_Startup**
Ein Zeiger auf eine weitere Struktur, genannt FileSysStartupMsg. Diese wird weiter unten besprochen.

***dn_GlobalVec**
Nur für BCPL-Programmierer interessant.

***dn_Name**
Der Name des Devices als BCPL-String, z.B. "DF0:".

Die DevList-Struktur für Disketten

Auch alle eingelegten Disketten werden über eine DevList-Struktur in die DOS-Base eingebunden. Hier die Struktur:

```
00      dc.l    *dl_Next              ; BCPL-Zeiger auf nächste Struktur
04      dc.l     dl_Type              ; Struktur-Typ (für Disks 2)
08      dc.l    *dl_Task              ; BCPL-Zeiger auf Handler-Task
12      dc.l    *dl_Lock              ; Für Disks nicht benutzt
16      ds.b     dl_VolumeDate,12     ; Erstellungsdatum (DateStamp!)
28      dc.l    *dl_LockList          ; BCPL-Zeiger auf Disk-Locks
32      dc.l     dl_DiskType          ; Diskettentyp (z.B. $444F5300)
36      dc.l     dl_unused            ; Nicht belegt
40      dc.l    *dl_Name              ; Diskname als BCPL-String
44               dl_SIZEOF
```

***dl_LockList**
Hier findet sich ein Zeiger auf die erste Struktur des ersten Locks, das auf die Diskette geholt wurde (über DOS-Routinen).

dl_DiskType
Dieser Eintrag entspricht dem ersten Langwort im Disk-Bootblock (Filesystem-Kennung).

12.2.5 Die FileSysStartupMsg-Struktur

Diese Struktur, die per Zeiger über die DeviceNode-Struktur erreicht werden kann, enthält erstens Informationen, die zum Öffnen des Devices per Exec-OpenDevice nötig sind. Zweitens sind die physikalischen Werte des Devices (Anzahl Tracks, Anzahl Sektoren usw.) abgelegt, was z.B. für die Untersuchung von Festplatteneinteilungen sehr nützlich ist.

```
00      dc.l     fssm_Unit            ; Exec-Unit-Nummer (für OpenDevice)
04      dc.l    *fssm_Device          ; Devicename als BCPL-String
08      dc.l    *fssm_Environ         ; BCPL-Zeiger auf Environment-Tab.
12      dc.l     fssm_Flags           ; Flags für OpenDevice
14               fssm_SIZEOF
```

Die Environment-Tabelle, auf die der Zeiger fssm_Environ weist, ist folgendermaßen aufgebaut ('de' steht für Disk Environment):

```
00      dc.l    de_TableSize          ; Anzahl Langworte in der Tabelle
04      dc.l    de_SizeBlock          ; Größe eines Blocks in Langworten
08      dc.l    de_SecOrg             ; Nicht benutzt; muß 0 sein
12      dc.l    de_NumHeads           ; Anzahl der Schreib/Lese-Köpfe
16      dc.l    de_SecsPerBlk         ; Nicht benutzt, muß 1 sein
20      dc.l    de_BlksPerTrack       ; Anzahl Blöcke auf einer Spur
24      dc.l    de_ReservedBlks       ; Anzahl Sektoren im Bootblock
28      dc.l    de_Prefac             ; Nicht benutzt, Muß 0 sein
32      dc.l    de_Interleave         ; Gewöhnlich 0
36      dc.l    de_LowCyl             ; Nummer des ersten Zylinders
40      dc.l    de_UpperCyl           ; Nummer des letzten Zylinders
44      dc.l    de_NumBuffers         ; Anzahl der Speicherpuffer
48      dc.l    de_MemBufType         ; Speichertyp für Puffer
52              de_SIZEOF
```

de_TableSize
Den ersten Eintrag nicht mitgezählt umfaßt die Tabelle 11 Langworte. Dies ist der Standardwert, der in der Regel im TableSize-Eintrag zu finden ist.

de_SizeBlock
Der Standardwert ist 128, welcher sich wahrscheinlich auch nie ändern wird.

de_NumHeads
Diskettenlaufwerke haben zwei Schreib/Lese-Köpfe, bei Festplatten z.B. kann der Wert variieren.

de_BlksPerTrack
Der Standardwert für Disketten ist 11. Abweichungen bei anderen Medien sind möglich.

de_ReservedBlks
Bei den Amiga-Filesystems umfaßt der Bootblock immer 2 Sektoren.

de_Interleave
Dieser Wert ist nur für Festplatten interessant.

de_LowCyl, de_HighCyl
Der erste bzw. letzte Zylinder auf dem Speichermedium, der für dieses "Gerät" benutzt wird. Bei Disketten gewöhnlich 0 bzw. 79, z.B. Festplatten können aber partitioniert, d.h. in mehrere logische Geräte eingeteilt werden, weshalb die Angabe des ersten und letzten Zylinders wichtig ist.

de_NumBuffers
Die Anzahl der jeweils 512 Byte umfassenden Speicherpuffer. Je größer die Pufferzahl, desto mehr Blöcke können im Speicher zwischengelagert werden.

de_MemBufType
Die Wertebelegung für diesen Eintrag entspricht den Speichertyp-Werten bei Aufruf von z.B. AllocMem. MemBufType gibt an, welcher Speichertyp für die Puffer des Gerätes nötig ist.

12.3 Programmstart mit der DOS-Library

Um einen unabhängigen Task zu starten, kennen wir bisher nur die AddTask-Funktion der Exec-Library. Doch auch die Dos-Library bietet uns die Möglichkeit, einen Task zu starten, und zwar mit der CreateProc-Funktion.

CreateProc	=	**-138 (DOS-Library)**

Name	d1	<	Zeiger auf den Namen, den der Prozess erhalten soll.
Pri	d2	<	Priorität, mit der der Prozess ausgestattet werden soll.
segList	d3	<	Zeiger auf die Segment-Liste (LoadSeg) des zu startenden Prozesses.
stackSize	d4	<	Größe des Stackbereichs, der angelegt werden soll.
Erklärung			Die CreateProc-Funktion legt einen durch die angegebenen Parameter bestimmten Prozess an.

Wie man sieht, benötigt die Funktion einen Zeiger auf eine Segment-Liste. Diese bekommen wir z.B., wenn ein Programm mittels LoadSeg geladen wurde. So könnten wir von unserem Programm aus ein zweites Programm einladen (LoadSeg) und mit CreatProc in Gang setzen.

Neben dieser recht "netten" Möglichkeit bietet sich noch etwas viel besseres an. So können wir z.B. das eigene Programm durch CreateProc von dem laufenden CLI-Task abhängen. Dies käme einem Aufruf mit RUN gleich. Dabei benutzen wir die Section-Anweisung, die das Programm in verschiedene Sectionen (Segmente) aufteilt. Das hat zur Folge, daß das Programm, wenn es mit LoadSeg geladen wurde, nicht in einem gemeinsamen Segment, sondern in zwei verschiedenen liegt.

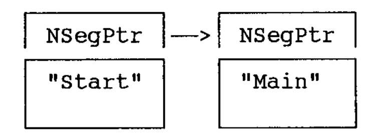
```
        ...
Start:
        move.l      ExecBase,a6
        ...

        Section     "",Code             ; hier wird das Programm geteilt

Main:
        move.l      ExecBase,a6
        ...
```

Bild 12.n: Unterteilung eines Programmes mittels SECTION

Wird dieses Programm in den Speicher geladen, wird es in zwei Segmente aufgeteilt, die nur durch einen BCPL-Zeiger



(Next-Segment-Pointer) verbunden sind. Um nun den eigenen Prozess zu starten, lesen wir den Zeiger auf das zweite Segment aus und übergeben ihn an die CreateProc-Funktion. Danach müssen wir den Zeiger mit Null überschreiben. Dies ist notwendig, damit beim Verlassen des ersten Segments nicht automatisch das zweite freigegeben wird.

Wie man sieht, ist es gar nicht so schwierig. Sollten trotzdem noch Probleme auftreten, wird hoffentlich das folgende Demonstrationsprogramm helfen.

```
*
* Kapitel 12
* Demonstrationsprogramm zu CreateProc
*

ExecBase       =       4
CreateProc     =       -138
OpenLib        =       -552
CloseLib       =       -414
Output         =       -60
Write          =       -48
Delay          =       -198
Close          =       -36
Open           =       -30


Start:
        move.l  ExecBase,a6     ; Dos-Library öffnen
        lea     DosName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)
        move.l  d0,a6
        beq     DosError
        move.l  d0,DosBase

        lea     Start-4,a0      ; Adresse des Pointer nach a0
        move.l  (a0),d3         ; BCPL-Zeiger auslesen
        clr.l   (a0)            ; und direkt löschen

        move.l  #Name,d1        ; nun wird der Prozess gestartet
        move.l  #-100,d2        ; Priorität
        move.l  #600,d4         ; Stack
        jsr     CreateProc(a6)  ; starten

        jsr     Output(a6)      ; Ausgabekanal ermitteln
        move.l  d0,d1
        move.l  #Text,d2
        move.l  #TextE-Text,d3
```

```
        jsr     Write(a6)       ; Text ausgeben

DosError:
        rts                     ; Ende


Text:           dc.b    10,"Prozess wurde gestartet!",10
                dc.b       "In fünf Sekunden erscheint das"
                dc.b       " Fenster!",10,10
TextE:          even

DosName:dc.b    "dos.library",0
                even


        Section ""，Code        ; <<<<< Trennung <<<<<<


Main:   move.l  DosBase,a6      ;
        move.l  #5*50,d1        ; fünf Sekunden warten
        jsr     Delay(a6)

        move.l  #Window,d1      ; Zeiger auf Fensterdaten
        move.l  #1005,d2        ; Modus
        jsr     Open(a6)        ; Fenster öffnen
        move.l  d0,-(a7)

        move.l  #10*50,d1       ; zehn Sekunden warten
        jsr     Delay(a6)

        move.l  (a7)+,d1
        jsr     Close(a6)       ; Fenster schließen

        move.l  ExecBase,a6
        move.l  DosBase,a1
        jsr     CloseLib(a6)    ; Dos-Lib schließen
        rts

Name:           dc.b    "Test-Prozess",0
                even
Window:         dc.b    "CON:50/80/540/40/Test-Prozess-Fenster (10sec)",0
                even
DosBase:        dc.l    0
```

Programm 12.2: Demonstration CreateProc

12.3.1 Die Prozess-Struktur

Noch etwas sollte man zu der Funktion CreateProc erwähnen. Sie erstellt nicht nur eine Task-Struktur für das neue Programm, sondern eine Process-Struktur. Dies ist eine etwas aufgepeppte Variante der Task-Struktur. Sie setzt sich aus einer Task- und MessagePort-Struktur und aus noch 16 weite-

ren Einträgen zusammen. (BP = BPTR = BCPL-Zeiger = APTR/4 = Adresse/4)

Process-Struktur:

```
000    dc.l    *tc_Succ              ;┐
004    dc.l    *tc_Pred              ;│
008    dc.b     tc_Type              ;├─ Node-Struktur
009    dc.b     tc_Pri               ;│
010    dc.l    *tc_Name              ;┘
014    dc.b     tc_Flags             ;
015    dc.b     tc_State             ;
016    dc.b     tc_IDNestCnt         ;
017    dc.b     tc_TDNestCnt         ;
018    dc.l     tc_SigAlloc          ;
022    dc.l     tc_SigWait           ;
026    dc.l     tc_SigRecvd          ; ├─ Task-Struktur
030    dc.l     tc_Except            ;      (pr_Task)
034    dc.w     tc_TrapAlloc         ;
036    dc.w     tc_TrapAble          ;
038    dc.l     tc_ExceptData        ;
042    dc.l     tc_ExceptCode        ;
046    dc.l     tc_TrapData          ;
050    dc.l     tc_TrapCode          ;
054    dc.l     tc_SPReg             ;
058    dc.l     tc_SPLower           ;
062    dc.l     tc_SPUpper           ;
066    dc.l    *tc_Switch            ;
070    dc.l    *tc_Launch            ;
074    dcb.b    tc_MemEntry,14       ;
088    dc.l     tc_UserData          ;┘

092    dc.l    *mp_Succ              ;┐
096    dc.l    *mp_Pred              ;│
100    dc.b     mp_Type              ;│
101    dc.b     mp_Pri               ;│
102    dc.l    *mp_Name              ;├─ MessagePort-Struktur
106    dc.b     mp_Flags             ;│       (pr_MsgPort)
107    dc.b     mp_SigBit            ;│
108    dc.l    *mpSigTask            ;│
112    dcb.b    mpMsgList,14         ;┘

126    dc.w     pr_Pad               ;    Füllwort
128    dc.l    *pr_SegList           ; BP Zeiger auf Segmentliste
132    dc.l     pr_StackSize         ;    Größe des Stacks
136    dc.l     pr_GlobVec           ;    globaler Vektor
140    dc.l     pr_TaskNum           ;    Nummer des CLI-Tasks
144    dc.l     pr_StackBase         ; BP obere Grenze des Stacks
148    dc.l     pr_Result2           ;    zweiter Rückgabewert
152    dc.l    *pr_CurrentDir        ; BP Zeiger auf Lock
156    dc.l    *pr_CIS               ; BP Zeiger auf Ausgabe-FileH
160    dc.l    *pr_COS               ; BP Zeiger auf Eingabe-FileH
164    dc.l    *pr_ConsoleTask       ;    Zeiger auf Console-Hd
168    dc.l    *pr_FileSystemTask    ;    Zeiger auf File-System-Task
172    dc.l    *pr_CLI               ; BP Zeiger auf CLI-Struktur
176    dc.l    *pr_RrturnAddr        ;    Zeiger auf End-Routine
```

```
180    dc.l    *pr_PktWait        ;    eigene "Wait"-Routine
184    dc.l    *pr_WindowPtr      ;    Zeiger auf Fenster
188             pr_SIZEOF
```

pr_Task
Da es sich bei einem Process eigentlich um einen Task handelt, beinhaltet die Process-Struktur natürlich auch eine Task-Struktur. Die Signal-Bits des Tasks sind zum Teil vorbestimmt.

Name	Bit #	Bedeutung
CTRL_C	12	Ctrl-C gedrückt
CTRL_D	13	Ctrl-D gedrückt
CTRL_E	14	Ctrl-E gedrückt
CTRL_F	15	Ctrl-F gedrückt

Mit Hilfe der gesetzten Bits kann man sehr einfach die Control-Kombinationen Ctrl-C, Ctrl-D, Ctrl-E und Ctrl-F überprüfen (näheres dazu gleich).

pr_MsgPort
In der Process-Struktur ist zusätzlich zu der Task-Struktur noch ein Message-Port eingebettet, der von DOS benutzt wird, um mit dem Task zu kommunizieren.

pr_Pad
Füllwort, um die folgenden Struktur-Daten auf Langwortgrenze zu bringen. Dies muß sein, da die Dos-Library, die auf die nachfolgenden Einträge zugreifen soll, mit der Programmiersprache BCPL erstellt wurde. BCPL kann aber lediglich durch vier teilbare Adressen ansprechen (BCPL-Pointer = Adresse/4).

***pr_SegList** (BCPL-Pointer!)
Zeiger auf eine BCPL-Tabelle, die die BCPL-Pointer der Segmente enthält.

```
*pr_SegList * 4 ———>   +-----------------------------+
                       |     Anzahl Zeiger (n)       |
                       | BCPL-Pointer auf Segment 1  |
                       | BCPL-Pointer auf Segment 2  |
                       | BCPL-Pointer auf Segment n  |
                       +-----------------------------+
```

pr_StackSize
Größe des Stackbereichs.

pr_GlobVec
Globaler Vektor für Prozesse.

pr_TaskNum
Nummer des CLI-Tasks, von dem dieser Process aufgerufen worden ist. Handelt es sich nicht um einen vom CLI gestarteten Process, ist der Eintrag mit Null initialisiert.

pr_StackBase (BCPL-Pointer!)
Adresse des oberen Stackbereichs des Tasks.

pr_Result2
Der pr_Result2-Eintrag ist dafür vorgesehen, Übergabewerte von aufgerufenen Funktionen zu bekommen, die als Fehlererkennung eine Null zurückliefern.

***pr_CurrentDir** (BCPL-Pointer!)
Zeiger auf die Lock-Struktur des aktuellen Verzeichnisses. Dieses Verzeichnis wird immer dann benutzt, wenn eine Datei geöffnet wird, bei der keine explizite Verzeichnisangabe übergeben worden ist.

***pr_CIS, *pr_COS** (BCPL-Pointer!)
Zeiger auf das FileHandel für die Aus- und Eingabeoperationen. Diese Einträge sind nur belegt wenn es sich um einen Process handelt, der vom CLI gestartet wurde.

***pr_ConsoleTask**
Zeiger auf den Handler, der für das Console-Fenster zuständig ist.

***pr_FileSystemTask**
Zeiger auf das Device, welches zur Zeit vom Task benutzt wird.

***pr_CLI** (BCPL-Pointer!)
Zeiger auf eine CommandLineInterpreter-Struktur (wird gleich noch erklärt). Dieser Wert ist initialisiert, wenn der Prozess vom CLI aufgerufen worden ist. Sonst steht hier eine Null.

***pr_ReturnAddr**
Zeiger auf die Rücksprungadresse, die auf dem Stack abgelegt ist.

***pr_PktWait**
Zeiger auf eine Routine, die ausgeführt werden soll, bevor der Process auf eine Meldung wartet. Setzt man keinen Zeiger ein, wird direkt die Standardroutine aufgerufen.

***pr_WindowPtr**
Der Eintrag *pr_WindowPtr hat Einfluß auf den Requester, der bei einer Fehlermeldung ausgegeben werden soll.

Wert	Erklärung
-1L ($FFFFFFFF)	Es wird kein Requester erstellt (der Task wird direkt über den aufgetretenen Fehler informiert).
0L ($00000000)	Der Requester wird auf der WorkBench erstellt.
***Eigenes Fenster**	Der Requester wird in dem angegebenen Fenster erstellt.

12.3.2 Die CLI-Struktur

Nachdem wir die Process-Struktur kennengelernt haben, wollen wir uns nun die angesprochene CommandLineInterface-Struktur ansehen.

CommandLineInterface-Struktur:

```
00     dc.l    cli_Result2          ;    Fehlernummer
04     dc.l   *cli_SetName          ; BP aktuelles Direktory
08     dc.l   *cli_CommandDir       ; BP Zeiger auf Command-Lock
12     dc.l    cli_ReturnCode       ;    FAILAT-Wert
16     dc.l   *cli_CommandName      ; BP Name des Befehls
20     dc.l    cli_FailLevel        ;    Fehlergrenze
24     dc.l   *cli_Prompt           ; BP Zeiger auf Prompt-Text
28     dc.l   *cli_StandardInput    ; BP Standard-Eingabekanal
32     dc.l   *cli_CurrentInput     ; BP umgeleitete Eingabe
36     dc.l   *cli_CommandFile      ; BP Name der Batchdatei
40     dc.l    cli_Interactive      ;    Interactive-Flag
44     dc.l    cli_Background       ;    Background-Flag
48     dc.l   *cli_CurrentOutput    ; BP umgeleitete Ausgabe
52     dc.l    cli_DefaultStack     ;    Default-Stackgröße
56     dc.l   *cli_StandardOutPut   ; BP Standard-Ausgabekanal
60     dc.l   *cli_Module           ; BP Zeiger auf 1. Segment
64             cli_SIZEOF
```

cli_Result2
Fehlernummer des letzten Fehlers.

***cli_SetName** (BCPL-Pointer!)
BCPL-Zeiger auf einen BCPL-String mit dem Namen des aktuellen Directory.

Zur Erinnerung des Aufbaus eines BCPL-Strings:

Länge	Zeichenkette

***cli_CommandDir** (BCPL-Pointer!)
Zeiger auf eine Lock-Struktur des Verzeichnisses, in dem nach dem angegebenem Befehl gesucht werden soll.

cli_ReturnCode
Der Entrag cli_ReturnCode enthält den Rückgabewert einer Routine, die durch den CLI-Befehl FAILAT überprüft wird. Dabei werden folgende Werte unterstützt:

Name	Wert	Bedeutung
RETURN_OK	0	kein Fehler
RETURN_WARN	5	Warnung
RETURN_ERROR	10	Fehler aufgetreten
RETURN_FAILT	20	totaler Fehler bzw. mehrere Fehler

***cli_CommandName** (BCPL-Pointer!)
BCPL-Zeiger auf einen BCPL-String, der den Namen des aufgerufenen Kommandos enthält.

cli_FailLevel
Durch den Wert cli_FailLevel wird festgelegt, durch welchen Fehler eine Meldung provoziert wird.

***cli_Prompt** (BCPL-Pointer!)
BCPL-Zeiger auf einen BCPL-String, der den Prompt-Text enthält.

***cli_StandardInput** (BCPL-Pointer!)
Zeiger auf den Standard-Eingabekanal, der zur Kommunikation benutzt werden soll.

***cli_CurrentInput** (BCPL-Pointer!)
Sollte der Befehl mit den Eingabe- (<) und Ausgabeumleitungszeichen (>) gestartet worden sein, steht hier der Zeiger auf den momentanen Eingabekanal (siehe auch cli_CurrentOutput).

***cli_CommandFile** (BCPL-Pointer!)
Ist der laufende Prozess von einer Batchdatei aufgerufen worden, wird hier der BCPL-Zeiger auf eine BCPL-Zeichenkette eingetragen, die den Namen der Batchdatei enthält.

cli_Interactive
Dieser Eintrag legt fest, ob das CLI interaktiv ist.

cli_Background
Auch der cli_Background-Eintrag ist ein Flag. Er gibt an, ob der CLI-Prozess im Hintergrund (Background) läuft.

***cli_CurrentOutput** (BCPL-Pointer!)
Zeiger auf den angegebenen Ausgabekanal, der momentan benutzt wird.

cli_DefaultStack
Der Wert von cli_DefaultStack gibt die Pflichtgröße des Stacks für den Task an.

***cli_StandardOutput** (BCPL-Pointer!)
Zeiger auf den Standard-Ausgabekanal, der zur Kommunikation benutzt werden soll.

***cli_Module** (BCPL-Pointer!)
Der Eintrag *cli_Module enthält die, durch vier geteilte Anfangsadresse des ersten Segments des gerade bearbeiteten Prozesses.

Zum Schluß dieses Kapitels möchte ich nochmals auf die schon belegten Signal-Bits für die Control-Kombinationen eingehen. Sie werden an unseren Task gesendet, wenn eine der Control-

Kombinationen gedrückt worden ist. Wie man sich das zunutze machen kann, soll das folgende Programm zeigen:

```
*
* Kapitel 12
* Demonstrationsprogramm für die CTRL-Überwachung
*

ExecBase        =       4
SetException    =       -312
OpenLib         =       -552
CloseLib        =       -414
Output          =       -60
Write           =       -48
Exit            =       -144


        move.l  ExecBase,a6

        lea     DosName,a1      ; Dos-Library öffnen
        moveq   #0,d0
        jsr     OpenLib(a6)
        move.l  d0,DosBase
        beq     DosError

        move.l  276(a6),a0      ; Zeiger auf laufenden Task "holen"
        move.l  #Exception,42(a0)       ; Zeiger auf Exception-Code
                                ; eintragen
        move.l  #%1000000000000,d0      ; Exception-Signal setzen (Ctrl-
C)
        move.l  d0,d1
        jsr     SetException(a6)

        move.l  DosBase,a6      ; Ausgabekanal ermitteln
        jsr     Output(a6)
        move.l  d0,d7

Loop:   move.l  d7,d1
        move.l  #Text,d2
        move.l  #TextE-Text,d3
        jsr     Write(a6)       ; Text ausgeben

        btst    #0,BranchFlag   ; Branchflag nicht gesetzt, dann
        beq     Loop            ; Schleife erneut durchlaufen

        move.l  ExecBase,a6
        move.l  DosBase,a1
        jsr     CloseLib(a6)    ; Library schließen

DosError:
        rts

Exception:
```

```
        cmp.l    #%1000000000000,d0    ; Wurde die Task-Exception durch
        bne      Restart          ; das Ctrl-C-Bit ausgelöst ?

        bset     #0,BranchFlag    ; Ja, dann BranchFlag setzen

Restart:rts

BranchFlag:   dc.w     0
DosBase:               dc.l      0
DosName:               dc.b      "dos.library",0
              even
Text:                  dc.b      "Bitte Ctrl-C drücken!",10
TextE:        even
```

Programm 12.3: Überwachung der CTRL-Tastenkombinationen

12.4 Kommunikation auf Dos-Ebene

Die Kommunikation auf Dos-Ebene ist eigentlich nur eine Spezialisierung des Message-Systems von Exec. So besteht das StandardPacket aus einer Message-Struktur und der DosPacket-Struktur.

StandardPacket-Struktur:

```
00      ds.b     sp_Msg,20          ; Message-Struktur
20      ds.b     sp_Pkt,48          ; DosPacket-Struktur
68               sp_SIZEOF
```

sp_Msg
Bei dieser Struktur handelt es sich um eine normale Exec-Message-Struktur, die die Grundlage für den Umgang mit den DosPackets bildet. Dabei wird jedoch der Eintrag mn_Name nicht als Zeiger auf eine Zeichenkette verstanden, sondern als Zeiger auf die DosPacket-Struktur, die mit ihr versendet werden soll.

sp_Pkt
Die zweite Struktur ist eine DosPacket-Struktur, deren Einträge gleich näher beschrieben werden.

Wie man sieht, besteht ein StandardPacket aus einer Message-Struktur, mit der die DosPacket-Struktur verschickt werden soll. In dieser DosPacket-Struktur ist das Kommando, sowie die Parameter abgelegt.

DosPacket-Struktur:

```
00      dc.l     *dp_Link           ; Zeiger auf die Message-Struktur
04      dc.l     *dp_Port           ; Zeiger auf ReplyPort
```

```
08      dc.l    dp_Type/dp_Action     ; Kommando
12      dc.l    dp_Res1/dp_Status     ;┐Rückgabewerte
16      dc.l    dp_Res2/dp_Status2    ;┘
20      dc.l    dp_Arg1/dp_BufAddr    ;┐
24      dc.l    dp_Arg2               ;│
28      dc.l    dp_Arg3               ;│Argumente
32      dc.l    dp_Arg4               ;│
36      dc.l    dp_Arg5               ;│
40      dc.l    dp_Arg6               ;│
44      dc.l    dp_Arg7               ;┘
48              dp_SIZEOF
```

***dp_Link**
In dp_Link ist die Adresse der Message-Struktur abgelegt, mit der die DosPacket-Struktur gesendet werden soll.

***dp_Port**
Der in dp_Port abgelegte Wert zeigt auf den ReplyPort, der benutzt werden soll.

dp_Type/dp_Action
Durch den Wert im Eintrag dp_Type (auch dp_Action genannt) wird der Befehl festgelegt, der ausgeführt werden soll. Dabei stehen folgende Befehle zur Verfügung.

Name	Wert	Bedeutung
ACTION_NIL	0	keine Aktion
ACTION_GET_BLOCK	2	Block von Diskette laden
ACTION_SET_MAP	4	SetMap
ACTION_DIE	5	Die
ACTION_EVENT	6	Event
ACTION_CURRENT_VOLUME	7	CurrentVolume
ACTION_LOCATE_OBJECT	8	LocateObject dp_Arg1 = Lock (BPTR) dp_Arg2 = Name (BSTR) dp_Arg3 = Modus dp_Res1 = Lock (BPTR)
ACTION_RENAME_DISK	9	RenameDisk dp_Arg1 = NewName (BPTR) dp_Res1 = TURE/FALSE
ACTION_WRITE	'W'	Write dp_Arg1 = FileHandle (BPTR) dp_Arg2 = Buffer dp_Arg3 = Length dp_Res1 = TRUE/FALSE
ACTION_READ	'R'	Read dp_Arg1 = FileHandle (BPTR) dp_Arg2 = Buffer dp_Arg3 = Length dp_Res1 = TRUE/FALSE
ACTION_FREE_LOCK	15	FreeLock dp_Arg1 = Lock (BPTR) dp_Res1 = TRUE/FALSE

ACTION_DELETE_OBJECT	16	DeleteObject
		dp_Arg1 = Lock (BPTR)
		dp_Arg2 = Name (BSTR)
		dp_Res1 = TRUE/FALSE
ACTION_RENAME_OBJECT	17	RenameObject
		dp_Arg1 = FromLock (BPTR)
		dp_Arg2 = FromName (BSTR)
		dp_Arg3 = ToLock (BPTR)
		dp_Arg4 = ToName (BSTR)
		dp_Res1 = TURE/FALSE
ACTION_MORE_CACHE	18	MoreCache
ACTION_COPY_DIR	19	CopyDir
		dp_Arg1 = Lock (BPTR)
		dp_Res1 = Lock (BPTR)
ACTION_WAIT_CHAR	20	WaitChar
		dp_Arg1 = Timeout
		dp_Res1 = TRUE/FALSE
ACTION_SET_PROTECT	21	SetProtect
		dp_Arg2 = Lock (BPTR)
		dp_Arg3 = Name (BSTR)
		dp_Arg4 = Maske
		dp_Res1 = TRUE/FALSE
ACTION_CREATE_DIR	22	CreateDir
		dp_Arg1 = Lock (BPTR)
		dp_Arg2 = Name (BSTR)
		dp_REs1 = Lock (BPTR)
ACTION_EXAMINE_OBJECT	23	Examine
		dp_Arg1 = Lock (BPTR)
		dp_Arg2 = FileInfoBlock (BPTR)
		dp_Res1 = TRUE/FALSE
ACTION_EXAMINE_NEXT	24	ExNext
		dp_Arg1 = Lock (BPTR)
		dp_Arg2 = FileInfoBlock (BPTR)
		dp_Res1 = TRUE/FALSE
ACTION_DISK_INFO	25	DiskInfo
		dp_Arg1 = InfoData (BPTR)
ACTION_INFO	26	Info
ACTION_FLUSH	27	Flush
ACTION_SET_COMMENT	28	SetComment
		dp_Arg2 = Lock (BPTR)
		dp_Arg3 = Name (BSTR)
		dp_Arg4 = Comment (BSTR)
		dp_Res1 = TRUE/FALSE
ACTION_PARENT	29	Parent
		dp_Arg1 = Lock (BPTR)
		dp_Res1 = ParentLock
ACTION_TIMER	30	Timer
ACTION_INHIBIT	31	Inhibit
		dp_Arg1 = TRUE/FALSE
		dp_Res1 = TRUE/FALSE
ACTION_DISK_TYPE	32	DiskType
ACTION_DISK_CHANGE	33	DiskChange
ACTION_SET_DATE	34	SetDate
ACTION_SCREEN_MODE	994	ScreenMode
ACTION_READ_RETURN	1001	ReadReturn
ACTION_WRITE_RETURN	1002	WriteReturn

```
ACTION_SEEK              1008 Seek
                              dp_Arg1 = FileHandle (BPTR)
                              dp_Arg2 = new rel. Position
                              dp_Arg3 = Modus
                              dp_Res1 = old Position
ACTION_FINDUPDATE        1004 FileUpDate
ACTION_FINDINPUT         1005 OpenOldFile
                              dp_Arg1 = FileHandle (BPTR)
                              dp_Arg2 = Lock (BPTR)
                              dp_Arg3 = Name (BSTR)
                              dp_Res1 = TRUE/FALSE
ACTION_FINDOUTPUT        1006 OpenNewFile
                              dp_Arg1 = FileHandle (BPTR)
                              dp_Arg2 = Lock (BPTR)
                              dp_Arg3 = Name (BSTR)
                              dp_Res1 = TRUE/FALSE
ACTION_END               1007 Close
                              dp_Arg1 = FileHandle (BPTR)
ACTION_TRUNCATE          1022 FFS only
ACTION_WRITE_PROTECT     1023 FFS only
```

dp_Res1/dp_Status, dp_Res2/dp_Status2,
Diese beiden Einträge enthalten, nachdem das "Packet" zurückgekommen ist, die Rückgabewerte.

dp_Arg1/dp_BufAddr, dp_Arg2, dp_Arg3, dp_Arg4, dp_Arg5, dp_Arg6, dp_Arg7
Die Einträge dp_Arg[1..7] müssen, bevor das "Packet" abgeschickt wird, mit den Übergabewerten belegt werden, die je nach Kommando unterschiedlich ausfallen können.

12.5 Parameterübergabe vom CLI

Ist ein Programm vom CLI aufgerufen worden, so erhält es nicht nur die Länge und die Adresse der übergebenen Parameterzeichenkette, sondern noch weitere Werte in den Adreßregistern 1 bis 6.

a0	Zeiger auf die übergebene Zeichenkette
a1	Adresse des Stacks
a2	Zeiger auf Dispatcher-Tabelle
a3	Größe des Stackbereichs
a4	Adresse des Programms
a5	Adresse der Dos-Dispatcher-Routine
a6	Rücksprungadresse
d0	Anzahl der übergebenen Zeichen

12.6 Parameterübergabe von der WorkBench

Nachdem wir uns mit der relativ einfachen Parameterübergabe bei CLI-Programmen beschäftigt haben, stellt sich die Frage, wie dies beim Aufruf durch ein Icon realisiert wird. Dabei benutzt man den, in der Process-Struktur eingerichteten Message-Port, an den uns die WorkBench eine Nachricht schickt. Diese Nachricht können wir, wie üblich, mit den Befehl GetMsg empfangen und dann auswerten.

Dabei erhält man eine Nachricht folgender Struktur:

WBStartup-Struktur:

```
00      ds.b     sm_Message,20        ; Message-Struktur
20      dc.l    *sm_Preocess          ; Zeiger auf Process-Descriptor
24      dc.l    *sm_Segment           ; Zeiger auf Segment-Descriptor
28      dc.l     sm_NumArgs           ; Anzahl der Elemente der ArgList
32      dc.l    *sm_ToolWindow        ; Zeiger auf Fenster-Descriptor
36      dc.l    *sm_ArgList           ; Zeiger auf Argumentenliste
40               sm_SIZEOF
```

Die Adresse, die im sm_ArgList-Eintrag abgespeichert worden ist, zeigt auf eine WBArg-Struktur, in der die eigentlichen Parameter abgelegt werden. Diese Parameter bestehen nicht aus einer Zeichenkette, sondern aus einem Zeiger auf die Lock-Struktur des zusätzlich aktivierten Icons (Shift+LMB) und dessen Namen.

WBArg-Struktur:

```
00      dc.l    *wa_Lock              ; Zeiger auf Lock
04      dc.l    *wa_Name              ; Zeiger auf Namen
08               wa_SIZEOF
```

12.7 Aufbau der ".info"-Datei

Die ".info"-Dateien kennen wir ja schon aus dem vorangegangenen Kapitel. Sie enthalten die Grafikdaten für das Icon, welches für eine bestimmte Datei erscheinen soll. Man kann sie mittels eines Icon-Editors anlegen oder sie mit einem kleinen Hilfsprogramm selbst zusammenbasteln. Dazu müssen wir uns zunächst den Aufbau einer solchen Datei ansehen.

12.7.1 Die DiskObject-Struktur

Jede ".info"-Datei fängt mit einer DiskObject-Struktur an:

DiskObject-Struktur:

```
00   dc.w    do_Magic            ; Erkennungsmarke
02   dc.w    do_Version          ; Version

04   dc.l   *gg_NextGadget       ;┐
08   dc.w    gg_LeftEdge         ;│
10   dc.w    gg_TopEdge          ;│
12   dc.w    gg_Width            ;│
14   dc.w    gg_Height           ;│
16   dc.w    gg_Flags            ;│
18   dc.w    gg_Activation       ;│ Gadget-Struktur
20   dc.w    gg_GadgetType       ;│
22   dc.l   *gg_GadgetRender     ;│
26   dc.l   *gg_SelectRender     ;│
30   dc.l   *gg_GadgetText       ;│
34   dc.l    gg_MutualExclude    ;│
38   dc.l   *gg_SpecialInfo      ;│
42   dc.w    gg_GadgetID         ;│
44   dc.l    gg_User             ;┘

48   dc.w    do_Type             ; DiskObject Typ
50   dc.l    do_DefaultTool      ; Standard-Tool
54   dc.l   *do_ToolTypes        ; Zeiger auf ToolTypes Einträge
58   dc.l    do_CurrentX         ; X-Position
62   dc.l    do_CurrentY         ; Y-Position
66   dc.l   *do_DrawerData       ; Zeiger auf DrawerData-Struktur
70   dc.l    do_ToolWindow       ; Fenster (Tool)
74   dc.l    do_StackSize        ; Größe des Stackbereichs (Tool)
78           do_SIZEOF
```

do_Magic
Die ".info"-Dateien müssen mit einem Erkennungswort beginnen.

```
WBDISK_MAGIC =        $e310
```

do_Version
Versionsnummer

do_Gadget
Gadget-Struktur, die benutzt wird, um das Icon zu beschreiben. Dabei werden nicht alle Einträge berücksichtigt.

do_Type
Angabe des Typs der ".info"-Datei. Folgende Typen stehen zur Auswahl:

Name	Wert	Bedeutung
WBDISK	1	Disketten-Icon
WBDRAWER	2	Unterverzeichnis-Icon
WBTOOL	3	Tool-Icon
WBPROJECT	4	Project-Icon
WBGARBAGE	5	Trashcan-Icon

do_DefaultTool
Zeiger auf Standard-Tool.

***do_ToolTypes**
Zeiger auf Tool-Types Einträge. Bei den Tool-Types-Einträgen handelt es sich um Zeichenketten, die nähere Informationen über das Programm geben können. Die Einträge der Tools werden wie folgt abgelegt.

```
dc.l   4 * (AnzahlEinträge + 1)
dc.l   Länge_der_Zeichenkette
dc.b   "Text..",0
dc.l   Länge_der_Zeichenkette
dc.b   "Text..",0
```

do_CurrentX, do_CurrentY
Momentane X- und Y-Position des Icons. Dabei gibt $80000000 an, daß dem Icon noch keine Position zugewiesen ist.

***do_DrawerData**
Zeiger auf DrawerData-Struktur, die benutzt wird, wenn es sich um Unterverzeichnisse handelt.

do_ToolWindow
Zeiger auf Standard-Fenster (nur bei Tools).

do_StackSize
Größe des Stackbereichs, der verwendet wird (nur bei Tools).

12.7.2 Die DrawerData-Struktur

Sollte es sich um eine ".info"-Datei des Typs "Drawer" handeln, so wird der Eintrag do_DrawerData mit einem Zeiger auf folgende Struktur versehen:

DrawerData-Struktur:

```
00    dc.w   nw_LeftEdge      ;┐
02    dc.w   nw_TopEdge       ;│
04    dc.w   nw_Width         ;│
06    dc.w   nw_Height        ;│
08    dc.b   nw_DetailPen     ;│
09    dc.b   nw_BlockPen      ;│
10    dc.l   nw_IDCMPFlags    ;│
14    dc.l   nw_Flag          ;│
18    dc.l   nw_FirstGadget   ;│ NewWindow-Struktur
22    dc.l   nw_CheckMark     ;│
26    dc.l   nw_Title         ;│
30    dc.l   nw_Screen        ;│
34    dc.l   nw_BitMap        ;│
38    dc.w   nw_MinWidth      ;│
40    dc.w   nw_MinHeight     ;│
42    dc.w   nw_MaxWidth      ;│
44    dc.w   nw_MaxHeight     ;│
46    dc.w   nw_Type          ;┘
```

```
48      dc.l    dd_CurrentX           ; X-Koordinate
52      dc.l    dd_CurrentY           ; Y-Koordinate
56              dd_SIZEOF
```

dd_NewWindow
Die DrawerData-Struktur besteht zum größten Teil aus einer NewWindow-Struktur, die das zu öffnende Fenster für das Unterverzeichnis beschreibt.

dd_CurrentX, dd_CurrentY
Momentane Position des Unterverzeichnisfensters.

12.7.3 Demonstrationsprogramm

Um nun eine ".info"-Datei zu erstellen, benötigen wir ein kleines Hilfsprogramm, welches die Daten direkt (ohne "Hunk"-Kennungen) speichert:

```
*
* Kapitel 12
* Demonstrationsprogramm zum Anlegen einer ".info"-Datei
*

ExecBase        =       4
Open            =       -30
Close           =       -36
Write           =       -48
OpenLib         =       -552
CloseLib        =       -414

        move.l  ExecBase,a6     ; Dos-Library öffnen
        lea     DosName,a1
        moveq   #0,d0
        jsr     OpenLib(a6)
        move.l  d0,DosBase
        move.l  d0,a6
        move.l  #Name,d1
        move.l  #1006,d2
        jsr     Open(a6)        ; Datei öffnen
        move.l  d0,Handle

        move.l  Handle,d1
        move.l  #Data,d2
        move.l  #DEnd-Data,d3
        jsr     Write(a6)       ; Daten speichern

        move.l  Handle,d1
        jsr     Close(a6)       ; Datei schließen
        move.l  DosBase,a1
        move.l  ExecBase,a6
        jsr     CloseLib(a6)    ; Dos-Library schließen
```

```
        rts

* Datenbereich

DosName:      dc.b    "dos.library",0
              even
Name:         dc.b    "ram:test.info",0
              even
DosBase:      dc.l    0
Handle:       dc.l    0

Data:
              ...     ; Daten der ".info"-Datei
DEnd:
```

Programm 12.4: Icon-Erstellung, Programm-Teil

Als nächstes müssen wir uns die Daten ansehen. Wir gehen davon aus, daß wir eine ".info"-Datei für das Unterverzeichnis "ram:test" erstellen wollen. Wichtig ist dabei, daß wir genau auf die Reihenfolge, DiskObject-Struktur, DrawerData-Struktur, Image-Struktur, Image-Data und ToolTypes-Einträge, achten.

```
Data:         dc.w    $e310          ; Erkennungsmarke
              dc.w    1              ; Versionsnummer

              dc.l    0              ; gg_NextGadget (unwichtig)
              dc.w    0              ; gg_LeftEdge
              dc.w    0              ; gg_TopEdge
              dc.w    128            ; gg_Width
              dc.w    60             ; gg_Height
              dc.w    6              ; gg_Flags
              dc.w    3              ; gg_Activation
              dc.w    1              ; gg_GadgetType
              dc.l    IData1         ; gg_GadgetRender (1.Image)
              dc.l    IData2         ; gg_SelectRender (2.Image)
              dc.l    0              ; gg_GadgetText (unwichtig)
              dc.l    0              ; gg_MutualExclude (unw.)
              dc.l    0              ; gg_SpecialInfo (unwichtig)
              dc.w    0              ; gg_GadgetID (unwichtig)
              dc.l    0              ; gg_User (unwichtig)

              dc.w    $200           ; do_Type = Drawer
              dc.l    0              ; do_DefaultTool
              dc.l    TTData         ; do_ToolTypes
              dc.l    $80000000      ; do_CurrentX, do_CurrentY
              dc.l    $80000000      ; (nicht bestimmt)
              dc.l    DrawerData     ; do_DrawerData
              dc.l    0              ; do_ToolWindow
              dc.l    0              ; do_StackSize
```

```
DrawerData:            ; dd_NewWindow
            dc.w   0,0              ; nw_LeftEdge/TopEdge
            dc.w   240,200          ; nw_Width/Height
            dc.b   0,0              ;┐
            dc.l   0,0,0,0,0,0,0,0       ;| unwichtig
            dc.w   0,0,0,0,0        ;|
            dc.l   0                ;┘

IData1:
            dc.w   0,0              ; ig_LeftEdge/TopEdge
            dc.w   128,60           ; ig_Width/Height
            dc.w   2                ; ig_Depth
            dc.l   ImageData1       ; ig_ImageData
            dc.b   %11,0            ; ig_PlanePick/PlaneOnOff
            dc.l   0                ; ig_NextImage

ImageData1:
            incbin PrgDisk:Kapitel_12/ImageData1.BM
                                    ; Image-Data einladen

IData2:
            dc.w   0,0              ; ig_LeftEdge/TopEdge
            dc.w   128,60           ; ig_Width/Height
            dc.w   2                ; ig_Depth
            dc.l   ImageData2       ; ig_ImageData
            dc.b   %11,0            ; ig_PlanePick/PlaneOnOff
            dc.l   0                ; ig_NextGadget

ImageData2:
            incbin PrgDisk:Kapitel_12/ImgaData2.BM
                                    ; Image-Data einladen

TTData:
            dc.l   4*3              ; (AnzahlEinträge + 1) * 4

            dc.l   TTData1E-TTData1
                                    ; Länge des Eintrags in Bytes
TTData1:
            dc.b   "ToolTypes 1",0
TTData1E:

            dc.l   TTData2E-TTData2    ;
                                    ; Länge des Eintrags in Bytes
TTData2:
            dc.b   "ToolTypes 2",0
TTData2E:

DEnd:
```

Programm 12.4: Icon-Erstellung, Daten-Teil

12.7 Aufbau einer "executable"-Datei

Nachdem wir uns von Anfang an damit beschäftigt haben, eigene Programme zu schreiben, ist es jetzt an der Zeit, uns anzusehen, wie diese Programme in Dateien abgelegt werden. Dazu gehen wir am besten von einem konkreten Beispiel aus.

Stellen wir uns vor, wir würden das folgende Programm assemblieren:

```
        move.l   #$4790,Data
        jmp      Ende

Data:        dc.l    0
Ende:        rts
```

Bild 12.n: Ein Test-Programm

Dabei hat uns Exec den Speicherbereich ab der Adresse $40000 zur Verfügung gestellt. Nach dem Assemblieren enthält dann der Speicherbereich $40000-$40014 folgende Daten:

Adresse	OpCode	; Programm
$40000	$23FC $0000 $4790 $0004 $0010	; move.l#$4790,Data
$4000A	$4EF9 $0004 $0014	; jmp Ende
$40010	$0000 $0000	; Data:
$40014	$4E75	; Ende: RTS

Wie man sieht, werden die absoluten Adressen für den Move-Befehl als auch für den Jump-Befehl direkt gespeichert. Würde man diesen Bereich nun in einer Datei ablegen und zu einem späteren Zeitpunkt wieder laden, könnte es sein, daß wir einen anderen Speicherbereich von Exec zugewiesen bekommen. Gehen wir davon aus, wir bekommen den Bereich von $50000-$50014.

Adresse	OpCode	; Programm
$50000	$23FC $0000 $4790 $0004 $0010	; move.l #$4790,Data
$5000A	$4EF9 $0004 $0014	; jmp Ende
$50010	$0000 $0000	; Data:
$50014	$4E75	; Ende: RTS

Wenn man das Programm jetzt abarbeitet, wird es mit hoher Wahrscheinlichkeit zu einem Absturz kommen. Dies liegt darin begründet, daß die absoluten Adressen nicht an die neue Anfangsadresse angepaßt worden sind. So würde der Jump-Befehl z.B. nach $40014 und nicht nach $50014 verzweigen.

Dieses Problem kann bei Monotasking-Betriebssystemen nicht auftreten, da die Programme immer an die gleiche Stelle geladen werden. Beim Amiga hingegen kann es sein, daß der Speicherbereich schon von einem anderen Programm oder anderen Daten belegt worden ist.

AmigaDos löst diesen Konflikt, indem die absoluten Adressen nach dem Laden an die neue Position angepaßt (relokiert) werden. Es werden dabei nicht die absoluten Adressen, sondern die relativen Offsets zum Anfang des Segments abgespeichert. Außerdem werden die Positionen der absoluten Adressen, die später wieder angepaßt werden sollen, in einer Offsettabelle abgelegt. Bei unserem Beispielprogramm sähe das wie folgt aus:

Adresse	OpCode	; Programm
$50000	$23FC $0000 $4790 $0000 $0010	; move.l #$4790,Data
$5000A	$4EF9 $0000 $0014	; jmp Ende
$50010	$0000 $0000	; Data:
$50014	$4E75	; Ende: RTS

```
Offsettabelle:
      dc.l       6
      dc.l       12
```

Nachdem das Programm geladen worden ist, wird die Anfangsadresse des Segments ($50000) zu den Offsetwerten addiert, die an Stelle der absoluten Adressen stehen. Dabei benutzt man die in der Offsettabelle abgelegten Werte, um an die anzupassende Stelle zu gelangen.

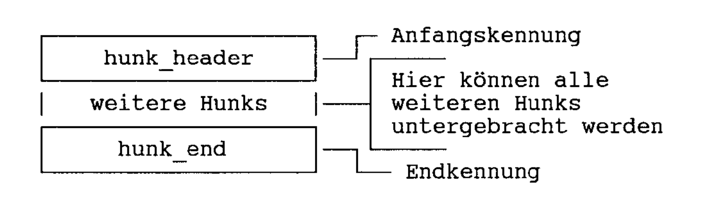
Neben der angesprochenen Offsettabelle werden natürlich noch weitere Daten abgelegt. Alle diese Datenteile haben einen eigenen Zahlenwert, mit dem man sie erkennen kann:

Name	Dec /Hex	Bedeutung
hunk_unit	999 /$03e7	Beginn einer Programmeinheit
hunk_name	1000/$03e8	Namenskennung
hunk_code	1001/$03e9	Kennung für ein Programmteil
hunk_data	1002/$03ea	Kennung für ein Datenteil
hunk_bss	1003/$03eb	Kennung des BSS-Teils
hunk_reloc32	1004/$03ec	Relokationstabelle (32-Bit)
hunk_reloc16	1005/$03ed	Relokationstabelle (16-Bit)
hunk_reloc8	1006/$03ee	Relokationstabelle (08-Bit)
hunk_ext	1007/$03ef	Externe Referenzen
hunk_symbol	1008/$03f0	Symbolteil
hunk_debug	1009/$03f1	Debuginformationen
hunk_end	1010/$03f2	Ende-Kennung
hunk_header	1011/$03f3	Anfangs-Kennung
hunk_overlay	1013/$03f5	Overlayblock
hunk_break	1014/$03f6	Overlay-Endkennung

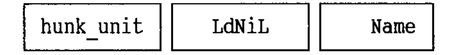
Folgende Abkürzungen werden verwendet:

LdNiL Länge des Namens in Langworten
LdCiL Länge des Codes in Langworten
LdDiL Länge der Daten in Langworten

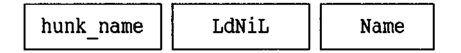
Eine ausführbare Datei muß immer folgenden Aufbau vorweisen:



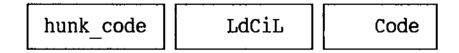
hunk_unit 999/$3e7
Durch die hunk_unit-Kennung wird ein Programm-Teil eingeleitet. Nach der Kennung folgt die Länge des Namens in Langworten und die Namensdaten selbst.



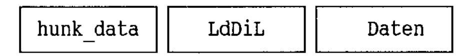
hunk_name 1000/$3e8
Der Name eines Hunks wird durch die Kennung hunk_name eingeleitet.



hunk_code 1001/$3e9
Der Hunk mit der Kennung hunk_code enthält Programmcode. Dabei wird nach der Kennung die Länge des Codes in Langworten erwartet.

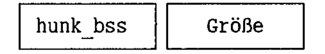


hunk_data 1002/$3ea
Vergleichbar mit hunk_code findet man bei hunk_data Daten mit bestimmten Werten (dc.x).

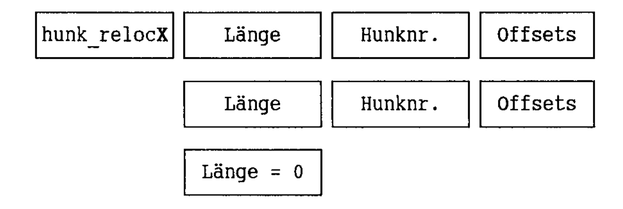


Auch ein Datenblock kann adreßabhängige Daten enthalten und deshalb auch mittels eines Relokationshunks angepaßt werden.

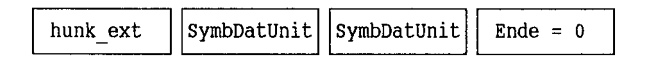
hunk_bss 1003/$3eb
Durch hunk_bss wird ein Hunk definiert, der nur aus zwei Langworten besteht. Nach der Kennung folgt die Größe des Blocks in Langworten, der für Daten zur Verfügung gestellt werden soll (ds.x).



hunk_reloc32, hunk_reloc16, hunk_reloc8 1004-1006/$3ec-$3ee
Durch diese drei Kennungen werden Hunks eingeleitet, die Daten zur Relokation enthalten. Diese Daten beziehen sich auf den davor liegenden Hunk. Nach der Kennung einer der Relokations-Hunks schließt sich die Länge der Offsetwerte an, die relokiert werden müssen. Außerdem folgt die Hunknummer, dessen Basisadresse dazu addiert werden soll, und die Offsets selbst. Dann kann sich wiederum ein Offsettabelle oder ein Null als Endkennung anschließen.



hunk_ext 1007/$3ef
Mit der Kennung hunk_ext wird ein Symbol-Hunk eingeleitet. Er besteht aus weiteren Symbol-Data-Units, wobei der Hunk letztlich durch ein Null-Langwort abgeschlossen wird.

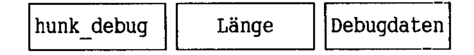


Jede dieser Symbol-Data-Units fängt wiederum mit einer Kennung an, welcher die Daten der Unit folgen.

Name:	Wert:	Bedeutung:
ext_symb	000	wird nur bei hunk_symb benutzt
ext_def	001	Definitionen (relokiert)
ext_abs	002	Definitionen (absolut)
ext_res	003	Definitionen residenter Libraries
ext_ref32	129	32-Bit Bezug auf Symbol
ext_ref16	131	16-Bit Bezug auf Symbol
ext_ref8	132	8-Bit Bezug auf Symbol
ext_common	130	32-Bit Bezug auf allgemeinen Block

hunk_symbol 1008/$3f0
Der symbol-Hunk hat grundsätzlich den gleichen Aufbau wie der ext-Hunk. Nach der Kennung folgen Symbol-Data-Units, die durch eine 0L abgeschlossen werden. Der Unterschied besteht darin, daß dieser Hunk lediglich von Debuggern und nicht, wie hunk_ext, vom Linker benutzt wird.

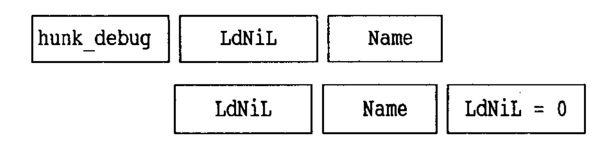
hunk_debug 1009/$3f1
In diesem Hunk können Daten abgelegt werden, die vom Debugger benutzt werden können. Dabei ist der Aufbau dieser Daten völlig frei. Lediglich die Länge des Hunks muß angegeben werden.



hunk_end 1010/$3f2
Diese Kennung, die nur aus einem Langwort besteht, beendet das Programm.

hunk_end

hunk_header 1011/$3f3
Durch die Kennung hunk_header wird der Anfang einer ausführbaren Datei gekennzeichnet. Nach der Kennung folgt ein Langwort, welches die Länge des nachfolgenden Namens in Langworten enthält, danach folgt der Name selbst. Nachdem die Namenskette mit Null abgeschlossen worden ist, schließt sich zunächst ein Langwort mit der Anzahl der Hunks an. Diesem Langwort folgt die Nummer des zuerst und zuletzt zu ladenden Hunks, sowie die Längenangaben aller Hunks.



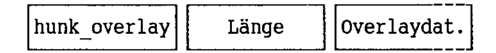
Anz.Hunks	erster Hunk	letzter H.	L.d.Hunks

Anz.Hunks = Anzahl aller Hunks + 1

L.d.Hunks = Länge der Hunks in Langworten

hunk_overlay 1013/$3f5

Die hunk_overlay-Kennung führt einen Overlay an. Durch das Overlay-Prinzip werden einzelne Teile des Programms, wenn sie benötigt werden, nachgeladen. Dabei wird kein neuer Speicherbereich benutzt, sondern ein Bereich, der schon durch das Programm belegt worden ist. Diese Auslagerungstechnik kann bei speicherintensiven Programmen benutzt werden.



hunk_break 1014/$3f6

Durch hunk_break wird ein Teil eines Overlay-Blocks abgeschlossen.

hunk_break

Anhang A
CLI-Schnellkurs

Das hierarchische Dateisystem

Allgemeines zu CLI-Befehlen

Physische und logische Geräte

Gerätezuweisung mit ASSIGN

Wichtige CLI-Befehle

Dieser Anhang stellt eine Kurzübersicht über die wichtigsten Funktionen des CLI, die Sie zur Assemblerprogrammierung benötigen, dar. Sie werden etwas über das Dateisystem, die Gerätebezeichnungen, die Pfadnamen und die wichtigsten Befehle erfahren. Außerdem finden Sie einige Tips zur Erstellung einer Arbeitsdiskette. Dies hier soll allerdings kein kompletter CLI-Kurs werden. Falls Sie noch mehr zum CLI erfahren möchten, empfehlen wir Ihnen die Lektüre des DOS-Handbuchs von Commodore etc.

Die Abkürzung "CLI" steht für "Command Line Interface", also "Kommandozeilen-Schnittstelle". Im Grunde kann das CLI nur zwei Dinge: Texte (Programmnamen) von der Tastatur einlesen und diese Programme ausführen. Sämtliche Befehle sind in Wirklichkeit Programmdateien auf der Diskette, die bei jeder Benutzung eingelesen werden müssen. Das erste, was man zum Umgang mit dem CLI kennen muß, ist also die Art und Weise, wie Programm- und Datendateien auf der Diskette verwaltet werden. "Diskette" ist hier übrigens ein Sammelbegriff für alle Speichermedien, also z.B. auch Fest- und Wechselplatten.

A.1 Das hierarchische Dateisystem

"Hierarchisches" Dateisystem bedeutet, daß eine gewisse "Rangordnung" im Diskettenaufbau vorherrscht. Da gibt es einmal die Diskette, die die höchste Instanz bildet. Dann kommen die Verzeichnisse, in denen schließlich die Dateien zu finden sind (dabei ist die Diskette eigentlich auch ein Verzeichnis, nämlich das Haupt-Verzeichnis). Die englische Bezeichnung "File" für eine Datei bedeutet eigentlich "Akte". Den Vergleich Datei-Akte kann man auch noch weiterführen: Die Diskette (das Hauptverzeichnis) ist mit einem Aktenschrank vergleichbar. Ein Verzeichnis ist eine Schublade in diesem Schrank. Ein Unterverzeichnis ist eine weitere Schublade in einer Schublade (was vielleicht nicht mehr so ganz vorstellbar ist). Eine Datei schließlich ist eine Mappe mit Akten.

Diese Mappe kann sich einmal im Aktenschrank befinden. Das trifft auf Dateien zu, die im Hauptverzeichnis einer Diskette stehen. Die Mappe kann sich auch in einer Schublade (in einem Verzeichnis) oder einer Unter-Schublade (in einem Unterverzeichnis) befinden. Wichtig ist, und da trifft unser Aktenschrank-Modell nicht mehr zu, daß Schubladen wieder beliebig viele Unter-Schubladen beinhalten können und diese wiederum Unter-Schubladen.

A.1.1 Datei- und Verzeichnisnamen

Lösen wir uns nun vom Aktenschrank-Modell und betrachten wir, wie man eine Amiga-Datei anspricht. Jede Datei bekommt zur Identifikation einen Namen. Dieser Name muß eindeutig sein, zwei Dateien gleichen Namens in einem Verzeichnis sind nicht möglich. Wohl aber dürfen in zwei verschiedenen Verzeichnissen Dateien gleichen Namens existieren, ja ein Verzeichnis darf sogar eine Datei beinhalten, die genauso heißt wie es selbst. Der Dateiname darf maximal 30 Zeichen lang sein und bis auf ":" und "/" alle druckbaren Zeichen beinhalten. Sollten allerdings Sonderzeichen wie das Leerzeichen, das Pluszeichen o.ä. im Namen enthalten sein, muß der gesamte Name in Anführungszeichen gesetzt sein. Für die Wahl von Verzeichnisnamen gelten dieselben Regeln.

A.1.2 Diskettennamen

Die Wahl von Diskettennamen erfolgt nach denselben Regeln wie die von Datei- und Verzeichnisnamen. Sie dürfen allerdings maximal 12 Zeichen lang sein. Im Anschluß an den Diskettennamen muß immer ein Doppelpunkt stehen. Man kann Disketten aber nicht nur unter ihrem Namen, sondern auch unter der Bezeichnung des Laufwerks, in dem sie sich befinden, ansprechen. Es gelten folgende Bezeichnungen:

DF0: - Das erste interne Disklaufwerk
DF1: - Das zweite Laufwerk (egal, ob intern oder extern)
DF2: - Das dritte Laufwerk (falls vorhanden)
DF3: - Das vierte Laufwerk
DH0: - Die Festplatte
DH1: - Die zweite Festplatte (erst mal eine haben ...)
RAM: - Die RAM-Disk (simulierte Diskette im Speicher)

Befindet sich z.B. eine Diskette namens "Assembler" im internen Laufwerk, kann man sie mit "DF0:" oder "Assembler:" ansprechen.

A.1.3 Pfadangaben

Angenommen, Sie haben auf der Diskette im Laufwerk DF0: ein Verzeichnis namens PROGRAMME, in diesem ein Unterverzeichnis namens ANLEITUNGEN und darin eine Datei namens TEST-ANL. Dann müssen Sie, um die Datei TEST-ANL eindeutig anzusprechen, folgende Angabe benutzen:

DF0:PROGRAMME/ANLEITUNGEN/TEST-ANL

Dieses Gebilde nennt sich "Pfadangabe" oder "Pfadname". Der Schrägstrich trennt in einer Pfadangabe immer den Namen eines Verzeichnisses von den Namen der untergeordneten Verzeichnisse oder Dateien. Anhand der kompletten Pfadangabe sucht sich das CLI seinen Weg durch die Dateistruktur der Diskette, angefangen im Hauptverzeichnis DF0:, dann ins

Verzeichnis PROGRAMME, von dort ins Unterverzeichnis ANLEITUNGEN, um dann die Datei TEST-ANL zu finden.

A.1.4 Setzen des aktuellen Verzeichnisses

Da die Eingabe des kompletten Pfadnamens bei Disketten mit vielen Unterverzeichnissen eine recht mühsame Sache sein kann, gibt es einen CLI-Befehl, der ein bestimmtes Verzeichnis zum aktuellen Verzeichnis erhebt. Er nennt sich "CD", das steht für "Current Directory". In obigem Beispiel könnte man nach Eingabe von

```
CD DF0:PROGRAMME/ANLEITUNGEN
```

die Datei TEST-ANL durch einfache Angabe ihres Namens erreichen. Die beiden Zeichen ":" und "/" haben in diesem Zusammenhang besondere Bedeutungen: Der Doppelpunkt steht für das Haupt-Verzeichnis der aktuellen Diskette, wenn keine Laufwerksangabe vor ihm steht. Der Schrägstrich bedeutet, wenn er ganz links in der Pfadangabe steht, daß auf das übergeordnete Verzeichnis zugegriffen werden soll. Ist also das Verzeichnis PROGRAMME/ANLEITUNGEN als das aktuelle gesetzt, und wollen Sie auf eine Datei namens TESTPRG im Verzeichnis PROGRAMME zugreifen, können Sie folgendermaßen verfahren:

```
/TESTPRG
```

Der Schrägstrich bedeutet, daß sich der folgende Dateiname auf das dem aktuellen Verzeichnis übergeordnete bezieht. Haben Sie eine Datei "TEST-2" im Hauptverzeichnis, können Sie so darauf zugreifen:

```
:TEST-2
```

Der Doppelpunkt bedeutet, daß vom Hauptverzeichnis der aktuellen Diskette ausgegangen werden soll. Ebenso wie es möglich ist, mit CD das aktuelle Verzeichnis zu wechseln, kann man auch das aktuelle Laufwerk wechseln:

```
CD DF1:
```

wechselt auf die Diskette, die im Laufwerk DF1: liegt. Wichtig: Der Doppelpunkt bezieht sich immer auf das Hauptverzeichnis des aktuellen Laufwerks, in diesem Beispiel wäre das also DF1:.

A.2 Allgemeines zu CLI-Befehlen

Wie schon erwähnt, sind alle CLI-Befehle in Wirklichkeit Programmdateien auf der Diskette. Wenn nun das CLI eine Eingabezeile bekommt, so gilt alles, was bis zum ersten Leerzeichen an Text eingegeben wird, als Befehl bzw. als

Dateiname. Der Rest der Eingabe (falls vorhanden) ist die sog. "Kommandozeile", auch "Parametertext" genannt, der dem zu startenden Programm übergeben wird. Ein Parameter ist die Angabe, auf welches Objekt (oder auf welche Objekte) sich der Befehl bezieht. Bei "CD DF0:" z.B. ist das CD der Befehlsname und das DF0: der Parametertext. Es gibt natürlich auch Befehle ohne Parameter. Der DIR-Befehl ohne Verzeichnisangabe z.B. gibt das aktuelle Verzeichnis aus.

Das CLI macht bei der Benennung von Dateien, Verzeichnissen und Disketten keinen Unterschied zwischen Groß- und Kleinschreibung. So bezeichnen "Test-Datei", "TEST-DATEI" und "tEsT-DaTeI" alle ein und dieselbe Datei. Bei der Auflistung von Verzeichnisinhalten wird allerdings die Schreibweise so aufgeführt, wie sie beim Anlegen der Datei angegeben wurde.

Es gibt die Möglichkeit, Ausgabetexte von Befehlen, die normalerweise ins CLI-Fenster gehen würden, in eine beliebige Datei umzuleiten. Dazu muß nach dem Befehlsnamen, abgetrennt durch ein Leerzeichen, ein ">" (Größerzeichen) folgen, und nach diesem der Name der Datei (eventuell inklusive Geräte- und Pfadangabe), in welche die Ausgabe gehen soll. Beispiel: Der DIR-Befehl gibt den Inhalt eines Verzeichnisses im CLI-Fenster aus. Soll die Ausgabe aber in die Datei "df0:VerzInhalt" gehen, schreibt man folgendes:

```
dir > df0:VerzInhalt df0:
```

Das zweite DF0: ist dann der eigentliche Parameter, hier das auszugebende Verzeichnis.

Des weiteren kann man durch Benutzung des "<" (Kleinerzeichens) eventuelle Eingaben, die ein Programm normalerweise von der Tastatur erwartet, aus einer Datei (oder von einem anderen Gerät) einlesen. Diese Möglichkeit wird aber nur sehr selten genutzt, daher wollen wir hier nicht näher darauf eingehen.

Bei der Vorstellung eines neuen CLI-Befehls gelten folgende Regeln für die Angabe der Syntax:

- CLI-Befehle und Schlüsselworte werden in GROSSBUCHSTABEN angegeben.
- Vom Anwender einzusetzende Parameter sind in gemischter Schreibweise.
- Wahlfreie Parameter (solche, die weggelassen werden können) stehen in eckigen Klammern.

A.3 Die Gerätebezeichnungen

Neben den Diskettenlaufwerken und Festplatten existieren noch einige weitere Geräte. Hier eine vollständige Liste der Geräte mit ihren Bezeichnungen:

Gerät	Beschreibung
DF0:-DF3:	Die maximal vier Diskettenlaufwerke
DH0:-DHn:	Festplatten
RAM:	Die normale RAM-Disk
RAD:	Die resetfeste RAM-Disk (ab Kickstart 1.3)
CON:	Konsolen-Fenster mit automatischer Editierung
RAW:	Konsolen-Fenster ohne automatische Editierung
NEWCON:	Eine verbesserte Version des CON-Gerätes (wird ab Kickstart 1.3 für die Shell verwendet)
SER:	Die serielle Schnittstelle
PAR:	Die parallele Schnittstelle
PRT:	Der in den Preferences eingestelle Drucker
SPEAK:	Das Sprachausgabe-System
NIL:	Scheingerät, das alle Ausgaben verwirft

A.3.1 Die beiden RAM-Disks RAM: und RAD:

Eine RAM-Disk ist eine simulierte Diskette im Speicher des Computers. Sie kann zur Zwischenspeicherung von häufig gebrauchten Befehlen und Dateien benutzt werden, da sie sehr hohe Zugriffsgeschwindigkeiten aufweist. CLI-Anwender, die nicht im Besitz einer Festplatte sind, können hier z.B. die CLI-Befehle ablegen, damit sie schneller geladen werden.

Der wichtigste Unterschied zwischen RAM: und RAD: ist der, daß der Inhalt von RAM: nach einem Neustart (Reset oder Einschalten) gelöscht ist, der von RAD: aber erhalten bleibt. Des weiteren arbeitet RAM: mit sog. "dynamischer" Speicherverwaltung, d.h. es nimmt immer so viel Speicherplatz ein, wie die vorhandenen Dateien benötigen. Das RAM: wird daher immer als zu 100% gefüllt angegeben. RAD: arbeitet mit "statischem" Speicher, es nimmt also immer so viel Platz ein, wie es maximal einnehmen kann (die Maximalgröße läßt sich einstellen).

Um RAM: benutzen zu können, muß lediglich die Datei "ram-handler" im L-Verzeichnis der Startdiskette stehen. Diese Datei, die Routinen für die RAM-Verwaltung enthält, wird beim ersten Zugriff auf RAM: automatisch geladen und bleibt bis zum nächsten Reset im Speicher.

Die RAD: ist hingegen ein Gerät, das nicht sofort beim Systemstart zur Verfügung steht. Es muß erst mit Hilfe des CLI-Befehls "MOUNT" angemeldet werden. Dazu wird die Datei "ramdrive.device" im DEVS-Verzeichnis benötigt. Näheres dazu erfahren Sie im nächsten Abschnitt.

A.3.2 Die Konsolen-Geräte CON:, RAW: und NEWCON:

Über diese Geräte hat man die Möglichkeit, Fenster zu öffnen und Eingaben aus diesen Fenstern entgegenzunehmen. Die Benutzung erfordert allerdings schon etwas Programmierwissen, weshalb wir uns im Kapitel 4 (über die DOS-Library) eingehend damit befaßt haben. An dieser Stelle nur soviel: Das Gerät NEWCON: muß erst angemeldet werden (Datei "newcon-handler" im L-Verzeichnis wird benötigt), während CON: und RAW: immer zur Verfügung stehen.

A.3.3 Schnittstellenansprache mit SER:, PAR: und PRT:

Diese drei Geräte dienen zum Senden von Daten an die serielle Schnittstelle (SER:), die parallele Schnittstelle (PAR:) und den im Preferences eingestellten Drucker (PRT:). Anstelle von PRT: können Sie auch SER: oder PAR: benutzen, je nachdem, an welcher Schnittstelle Ihr Drucker angeschlossen ist. So etwas wie eine Pfadangabe gibt es bei diesen Geräten natürlich nicht (wie sollte der Drucker auch Unterverzeichnisse haben). Man spricht sie einfach über die Gerätebezeichnung an. Um beispielsweise eine (Text-)Datei auszudrucken, kann man die folgenden Befehle benutzen:

```
COPY Datei TO PRT:   oder
TYPE > PRT: Datei
```

A.3.4 Say it again, Amiga - Die Sprachausgabe

Wenn Sie sich von Ihrem Amiga einmal so richtig "vollquatschen" lassen wollen, können Sie das Gerät "SPEAK:" benutzen. Es muß, genau wie RAD: und NEWCON:, über MOUNT angemeldet werden. Benötigt werden die Dateien "speak-handler" im L-Verzeichnis und "narrator.device" im DEVS-Verzeichnis. Die Ansteuerung erfolgt exakt genauso wie bei SER:, PAR: und PRT:. Mit folgenden Befehlen kann man sich eine Datei "vorlesen" lassen:

```
COPY Datei TO SPEAK:   oder
TYPE > SPEAK: Datei
```

A.3.5 NIL: - Das Nimmerland-Gerät

Das NIL steht für "Nichts", und dementsprechend verwirft dieses Gerät einfach alles, was an es gesendet wird. In einem Datei-Archivierungs-Programm namens "Zoo" wird das sehr treffend "send data to neverland" - "schicke Daten ins Nimmerland" genannt (Peter Pan läßt grüßen).

Dieses Gerät dient dazu, unerwünschte Textausgaben von CLI-Befehlen zu unterdrücken. Man braucht die Textausgabe lediglich mittels des Größerzeichens nach dem Befehlsnamen nach NIL: umzuleiten. Ein Anwendungsbeispiel: Der SetClock-Befehl

lädt die Systemzeit aus der Echtzeituhr. Anschließend gibt er die gerade gelesene Uhrzeit auf dem Bildschirm aus. Wenn Sie nun anstatt "SETCLOCK LOAD" "SETCLOCK > NIL: LOAD" schreiben, wird die Ausgabe unterdrückt.

A.3.6 Der MOUNT-Befehl - Anmeldung von Geräten

Manche Geräte stehen nicht sofort bei Systemstart zur Verfügung, sondern müssen erst angemeldet werden. Dazu zählt die resetfeste RAM-Disk RAD:, der verbesserte Konsolen-Handler NEWCON: und die Sprachausgabe SPEAK:. Um die RAD: anzumelden, genügt der Befehl

MOUNT RAD:

Der Mount-Befehl arbeitet über eine Text-Datei namens Mount-List, die sich im DEVS-Verzeichnis der Startdiskette befindet. Für jedes Gerät, das per MOUNT angemeldet werden kann, findet sich dort ein Eintrag. Wir stellen hier die für RAD:, NEWCON: und SPEAK: nötigen Einträge vor, ohne allerdings im Detail auf die einzelnen Schlüsselworte einzugehen. Weiterführende Informationen zum Thema MOUNT-Befehl finden Sie im Commodore-DOS-Handbuch.

```
RAD:         Device         = ramdrive.device
             Unit           = 0
             Flags          = 0
             Surfaces       = 2
             BlocksPerTrack = 11
             Reserved       = 2
             Interleave     = 0
             LowCyl         = 0
             HighCyl        = 10
             Buffers        = 20
             BufMemType     = 1
             Mount          = 1
#
```

Bild A.1: Der Mountlist-Eintrag für die RAD:

Sie können die Größe der RAD: verändern, indem Sie einen anderen Wert für HighCyl eingeben. Die Größe in Bytes beträgt HighCyl-Wert * 22528.

```
NEWCON: Handler  = L:Newcon-Handler
        Priority     = 5
        StackSize    = 1000
#
```

Bild A.2: Der Mountlist-Eintrag für NEWCON:

```
SPEAK:          Handler       = L:Speak-Handler
                Priority      = 5
                StackSize     = 6000
                GlobVec       = -1
#
```

Bild A.3: Der Mountlist-Eintrag für SPEAK:

A.4 Die logischen Geräte

Neben den "echten" Geräten, die wir bis jetzt kennengelernt haben, gibt es noch eine andere Art von Geräten: die logischen Geräte. Dabei handelt es sich um Verzeichnisse (oder auch Dateien), denen Gerätenamen zugewiesen werden. Beim Systemstart werden automatisch folgende Zuweisungen, die sich alle auf die Diskette beziehen, von der gestartet wird, vorgenommen:

Gerät	Beschreibung
SYS:	Hauptverzeichnis
C:	C-Verzeichnis (hier werden die CLI-Befehle gesucht)
DEVS:	DEVS-Verzeichnis (beinhaltet die verschiedenen Gerätetreiber)
FONTS:	FONTS-Verzeichnis (dürfte wohl klar sein, was es beinhaltet)
L:	L-Verzeichnis (beinhaltet System-Programme, die bei Bedarf geladen werden)
LIBS:	LIBS-Verzeichnis (beinhaltet alle nicht im ROM stehenden Libraries)
S:	S-Verzeichnis (hier werden Batch-Dateien vom CLI-Befehl 'Execute' automatisch gesucht)
T:	T-Verzeichnis (hier legen diverse Programme ihre temporären Arbeitsdateien an)

A.4.1 Geräte-Zuweisungen mit dem ASSIGN-Befehl

Ansehen und verändern kann man diese Zuweisungen mit dem ASSIGN-Befehl. Ohne Angabe von Parametern, listet dieser alle existierenden Zuweisungen auf. Ansonsten gilt folgendes Format:

```
ASSIGN Dev: Verz
```

Dem logischen Gerät 'Dev:' wird das Verzeichnis 'Verz' zugewiesen. Gewöhnlich werden neue Zuweisungen nur benötigt, wenn Programme, die z.B. in ein Verzeichnis auf einer Festplatte kopiert wurden, bestimmte Disketten verlangen. Angenommen, Sie haben ein Textprogramm namens "WordMaster" in ein Verzeichnis namens "WM" auf Ihrer Festplatte kopiert,

das Programm aber verlangt immer wieder mit "Insert volume WordMaster in any drive" das Einlegen der Programmdiskette (z.B. weil irgendwelche Daten nachgeladen werden müssen). Dann können Sie den Computer mit dem Befehl

ASSIGN WordMaster: DH0:WM

dazu bringen, das "Gerät" (also die Diskette) WordMaster unter dem Verzeichnis DH0:WM anzusprechen.

Ein weiteres Einsatzgebiet von ASSIGN ist die Änderung der System-Zuweisungen für C:, L: usw. Wenn Sie z.B. die CLI-Befehle in ein Verzeichnis namens "CLI" in der RAM-Disk kopiert haben und den Computer anweisen wollen, alle CLI-Befehle ab jetzt im RAM zu suchen, können Sie das so tun:

ASSIGN C: RAM:CLI

Analog können Sie auch die Standard-Suchverzeichnisse für Libraries, Fonts, Devices usw. verändern.

A.5 Sonstige wichtige CLI-Befehle

Nun eine Zusammenstellung der wichtigsten Befehle, die man im täglichen Umgang mit dem CLI braucht. Da dies hier allerdings kein CLI-Kurs werden soll, werden wir die Befehle nur kurz vorstellen. Für weitere Informationen und Beispiele lesen Sie bitte im DOS-Handbuch nach.

A.5.1 ADDBUFFERS

Syntax: ADDBUFFERS Laufw Anz

Dieser Befehl erhöht die Anzahl der Pufferspeicher für ein Laufwerk. Je mehr Pufferspeicher vorhanden ist, desto mehr Daten können zwischengelagert werden, bevor ein erneuter Diskzugriff nötig ist, was die Diskgeschwindigkeit spürbar erhöhen kann. Jeder Puffer belegt 512 Bytes im Speicher. Am günstigsten ist die Erhöhung der Pufferanzahl um 25. Ein noch größerer Speicher bringt kaum noch Zeitvorteile.

A.5.2 ASSIGN

Syntax: ASSIGN [Laufw [Verz]]

Assign ohne Parameter listet alle bestehenden Zuweisungen auf. Falls nur ein Gerätename angegeben wird, das Verzeichnis aber fehlt, wird die Zuweisung aufgehoben. Weitere Informationen finden Sie im Abschnitt "Die logischen Geräte".

A.5.3 COPY

Syntax: COPY [Quelle] TO Ziel [ALL] [QUIET]

Der Datei-Kopier-Befehl. Quelle und Ziel können sowohl Verzeichnisse als auch Dateien sein. Wenn Quelle fehlt, werden alle Dateien des aktuellen Verzeichnisses kopiert. Ziel sollte dann ein Verzeichnis sein. ALL bewirkt, daß auch alle Unterverzeichnisse des angegebenen Quellverzeichnisses mitkopiert werden. QUIET unterdrückt die Ausgabe der Dateinamen, die gerade kopiert werden.

A.5.4 DELETE

Syntax: DELETE Name [Name...] [ALL] [QUIET]

Löscht Dateien und Verzeichnisse. Verzeichnisse können nur gelöscht werden, wenn sie leer sind. ALL bewirkt das Löschen aller Unterverzeichnisse des angegebenen Verzeichnisses. QUIET unterdrückt die Bildschirmausgabe.

A.5.5 DIR

Syntax: DIR Name [OPT A] [OPT I]

Zeigt das angegebene Verzeichnis an. OPT A bewirkt die Ausgabe aller Unterverzeichnisse, und OPT I leitet den interaktiven Modus ein. Dabei haben Sie nach jedem ausgegebenen Verzeichniseintrag die Möglichkeit, durch Eingabe eines Kennbuchstabens zu entscheiden, wie es weitergehen soll. Diese Buchstaben sind:

- Q - Beendet die Verzeichnisausgabe.
- E - Wenn der derzeitige Eintrag ein Unterverzeichnis ist, können Sie es mit E öffnen und durchgehen.
- B - Wechselt aus einem Unterverzeichnis in das übergeordnete Verzeichnis zurück.
- DEL - Löscht den aktuellen Eintrag (falls es eine Datei oder ein leeres Verzeichnis ist).

A.5.6 ENDCLI

Syntax: ENDCLI

Beendet den aktuellen CLI-Prozeß.

A.5.7 EXECUTE

Syntax: EXECUTE Stapeldatei [Argumente]

Interpretiert jeweils eine Zeile der Stapeldatei als CLI-Befehl und führt diese nacheinander aus. Wenn der Dateiname

keine Pfadangabe hat, wird automatisch im S-Verzeichnis gesucht. Die Datei "STARTUP-SEQUENCE" im S-Verzeichnis wird immer automatisch bei Systemstart ausgeführt.

A.5.8 FORMAT

Syntax: FORMAT DRIVE Laufw NAME Diskname [NOICONS] [QUICK]

Formatiert eine Diskette (oder auch Festplatte). NOICONS verhindert das Anlegen des Trashcan auf den Diskette. QUICK löscht nur das Hauptverzeichnis, was sehr viel schneller geht als die normale Formatierung, aber nur Sinn hat, wenn die Diskette schon einmal formatiert wurde und nur schnell gelöscht werden soll.

A.5.9 INFO

Syntax: INFO

Gibt Informationen über alle eingelegten Disketten in jedem angeschlossenen Laufwerk aus.

A.5.10 MAKEDIR

Syntax: MAKEDIR Verzeichnis

Erstellt ein neues Unterverzeichnis. Die Angabe des neuen Verzeichnisses bezieht sich, falls kein kompletter Pfadname angegeben wird, auf das aktuelle Verzeichnis.

A.5.11 NEWCLI

Syntax: NEWCLI

Öffnet ein neues CLI-Fenster.

A.5.12 RELABEL

Syntax: RELABEL [DRIVE] Laufw [NAME] Name

Gibt der Diskette im Laufwerk 'Laufw' den neuen Namen 'Name'. Die Schlüsselworte DRIVE und NAME können wegfallen.

A.5.13 RENAME

Syntax: RENAME [FROM] Alt [TO] Neu

Benennt eine Datei oder ein Verzeichnis um. Dabei ist auch eine Verschiebung in ein anderes Unterverzeichnis möglich,

wenn komplette Pfadnamen angegeben werden. Die Verschiebung auf eine andere Diskette ist allerdings nicht möglich.

A.5.14 RUN

Syntax: RUN Befehl

Führt den angegebenen Befehl im Hintergrund aus. Falls das letzte Zeichen im Befehl ein "+" ist, kann ein weiterer Befehl eingegeben werden, der dann auch im Hintergrund ausgeführt wird. Die Ausführung beginnt erst, wenn ein Befehl ohne "+" am Ende eingegeben wird.

A.5.15 SETMAP

Syntax: SETMAP Tastaturbelegung

Stellt eine länderspezifische Tastaturbelegung ein. Die wichtigesten Kürzel sind 'D' für die deutsche Tastatur, 'GB' für die englische und 'USA0' für die amerikanische Standard-Belegung. Die entsprechenden Dateien müssen im DEVS/KEYMAPS-Verzeichnis stehen.

A.5.16 TYPE

Syntax: TYPE Datei [TO Ziel] [OPT N] [OPT H]

Gibt eine Textdatei auf dem Bildschirm oder, wenn TO Ziel angegeben ist, in eine Datei (oder auch auf dem Drucker) aus. OPT N bewirkt die Numerierung der Zeilen und OPT H eine Ausgabe als Hex-Dump.

Anhang B
Literaturverzeichnis

Literaturverzeichnis
(Alle Angaben ohne Gewähr)

AMIGA Hardware Reference Manual
Addison-Wesley (Commodore-Amiga Incl.)
512 Seiten englisch (ca. 69,- DM)
ISBN 0-201-56776-8

AMIGA Hardware Reference Manual (Devices)
Addison-Wesley (Commodore-Amiga Incl.)
512 Seiten englisch (ca. 69,- DM)
ISBN 0-201-56775-?

AMIGA Hardware Reference Manual (Libraries)
Addison-Wesley (Commodore-Amiga Incl.)
960 Seiten englisch (ca. 99,- DM)
ISBN 3-201-56774-1

AMIGA Hardware Reference Manual (Includes and Autodocs)
Addison-Wesley (Commodore-Amiga Incl.)
1000 Seiten englisch (ca. 99,- DM)
ISBN 0-201-56773-3

Amiga-OS/2 2.0 für Insider-Applikationsentwicklung
Markt & Technik (Gzella, H./Häring, W.)
400 Seiten (ca. 69,- DM)
ISBN 3-87791-273-7

Amiga-Programmierrichtlinien
Markt & Technik (Gzella, H.)
258 Seiten (ca. 49,- DM)
ISBN 3-87791-049-1

Amiga, Kommentiertes ROM-Listing Bd. 1
Mediscript (Ruprecht, O.)
269 Seiten (ca. 69,- DM)
ISBN 3-88320-168-5

Amiga, Kommentiertes ROM-Listing Bd. 2
Mediscript (Ruprecht, O.)
340 Seiten (ca. 69,- DM)
ISBN 3-88320-169-3

Amiga, Kommentiertes ROM-Listing Bd. 3
Mediscript (Ruprecht, O.)
586 Seiten (ca. 79,- DM)
ISBN 3-88320-184-7

Anhang C
Befehlsliste des MC68000

```
ABCD    Dx,Dy            Addiere Dezimal mit X-Flag
ABCD    -(Ax),-(Ay)

ADD.x   <ea>,Dn          Addiere binär
ADD.x   Dn,<ea>
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm
D: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ADDA.W <ea>,An           Addiere Adresse
ADDA.L <ea>,An
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

ADDI.x #Konst,<ea>
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ADDQ.x #Konst,<ea>
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ADDX.x Dx,Dy             Addiere mit X-Flag
ADDX.x -(Ax),-(Ay)

AND.x   <ea>,Dn          Und-Verknüpfung
AND.x   Dn,<ea>
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm
D: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ANDI.x #Konst,<ea>       Und-Verknüpfung mit Konstaten
D: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ANDI    #Konst,CCR       Und-Verknüpfung mit demBedingungscode-Register

ANDI    #Konst,SR        Und-Verknüpfung mit Statusregiter

ASL.x   Dx,Dy            Arithmetisch schieben
ASL.x   #Konst,Dn
ASL     <ea>
ASR.x   Dx,Dy
ASR.x   #Konst,Dn
ASR     <ea>
D: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

Bcc     Lab              bedingter Sprung
Bcc.S   Lab

BCHG    Dn,<ea>          Prüfe ein Bit und wechsle
BCHG    #Konst,<ea>
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

BCLR    Dn,<ea>          Prüfe ein Bit und lösche
BCLR    #Konst,<ea>
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

BRA     Lab              Springe immer
BRA.S   Lab
```

```
BSET    Dn,<ea>            Prüfe ein Bit und setze
BSET    #Konst,<ea>
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

BSR     Lab                Springe ins Unterprogramm
BSR.S   Lab

BTST    Dn,<ea>            Prüfe ein Bit
BTST    Dn,#Konst
BTST    #Konst,<ea>
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi)

CHK     <ea>,Dn            Prüfe Registerinhalte gegen Grenzen
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

CLR.x   <ea>               Lösche ein Operand
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

CMP.x   <ea>,Dn            Vergleiche
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

CMPA.x <ea>,An             Vergleiche Adresse
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

CMPI.   #Konst,<ea>        Vergleiche nachfolgend
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

CMPM.x (Ax)+,(Ay)+         Vergleiche Speicher

DBcc    Dn,Lab             Prüfe Bedingung und springe
DBRA    Dn,Lab

DIVS    <ea>,Dn            Vorzeichenbehaftete Division
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

DIVU    <ea>,Dn            Vorzeichenlose Division
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm


EOR.x   Dn,<ea>            Ausschließliche Oder-Verknüpfung
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

EORI.x #Konst,<ea>         Verknüpfe ausschließlich mit nachfolgendem Wert
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

EORI    #Konst,CCR         Verknüpfe ausschließlich mit Bedingungsregister

EORI    #Konst,SR          Verknüpfe ausschließlich mit dem Status-Register

EXG     Rx,Ry              Vertausche Registerinhalte

EXT.x   Dn                 Erweitere Vorzeichen

ILLEGAL                    Unerlaubter Befehl
```

```
JMP     <ea>              Springe
S: (An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi)

JSR     <ea>              Springe ins Unterprogramm
S: (An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi)

LEA     <ea>,An           Lade Adresse
S: (An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi)

LINK    An,#Konst         Binde und reserviere

LSL.x   Dx,Dy             Logisches Schieben
LSL.x   #Konst,Dn
LSL.W   <ea>
LSR.x   Dx,Dy
LSR.x   #Konst,Dn
LSR.W   <ea>
S: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

MOVE.x <ea>,<ea>          Verschiebe Daten von Quelle nach Ziel
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

MOVEA.x<ea>,An            Verschiebe Adresse
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

MOVE    <ea>,CCR          Verschiebe ins CC-Register
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

MOVE    <ea>,SR           Verschiebe ins Status-Register
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

MOVE    USP,An            Verschiebe USP/A7
MOVE    An,USP

MOVEM.LRegList,<ea>       Verschiebe mehrere Registerinhalte
MOVEM.L<ea>,RegList
S: (An),(An)+,d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi)
D: (An),-(An),d(An),d(An,Xi),Abs.W,Abs.L

MOVEP.xDx,d(Ay)           Verschiebe Peripherie-Daten
MOVEP.xd(Ax),Dy

MOVEQ   #Konst,Dn         Verschiebe schnell

MULS    <ea>,Dn           Vorzeichenbehaftete Multiplikation
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

MULU    <ea>,Dn           Vorzeichenlose Multiplikation
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

NBCD    <ea>              Negiere dezimal mit Erweiterung
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

NEG.x   <ea>              Negiere
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L
```

```
NEGX.x <ea>             Negiere mit Erweiterung
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

NOP                     Keine Operation

NOT.x  <ea>             Logische Komplimentierung
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

OR.x   <ea>,Dn          Einschließliche Oder-Verknüpung
OR.x   Dn,<ea>
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm
D: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ORI.x  #Konst,<ea>      Einschließliche Oder-Verknüpfung nachfolgend
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ORI    #Konst,CCR       Einschließliche Verknüpfung mit CC-Register

ORI    #Konst,SR        Einschließliche Verknüpfung mit Status-Register

PEA    <ea>             Adresse auf Stack ablegen
D: (An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi)

RESET                   Externe Einheiten zurücksetzen

ROL.x  Dx,Dy            Rotiere (ohne Erweiterung)
ROL.x  #Konst,Dx
ROL.W  <ea>
ROR.x  Dx,Dy
ROR.x  #Konst,Dx
ROR.W  <ea>
S: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

ROXL.x Dx,Dy            Rotiere mit Erweiterung
ROXL.x #Konst,Dx
ROXL.W <ea>
ROXR.x Dx,Dy
ROXR.x #Konst,Dx
ROXR.W <ea>
S: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

RTE                     Rückkehr von Ausnahmezustand

RTR                     Kehre zurück und lade CCR

RTS                     Kehre vom Unterprogramm zurück

SBCD   Dy,Dx            Subtrahiere dezimal mit Erweiterung
SBCD   -(Ax),-(Ay)

SCC    <ea>             Setze, je nach CCR
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

STOP   #Konst           Lade Status-Register und halte
```

```
SUB.x   <ea>,Dn             Subtrahiere binär
SUB.x   Dn,<ea>
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm
D: (An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

SUBA.W <ea>,An              Subtrahiere Adresse
SUBA.L <ea>,An
S: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L,d(PC),d(PC,Xi),#Imm

SUBI.x #Konst,<ea>          Subtrahiere nachfolgend
D: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

SUBQ.x #Konst,<ea>          Subtrahiere schnell
D: Dn,An,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

SUBX.x Dx,Dy                Subtrahiere mit Erweiterung
SUBX.x -(Ax),-(Ay)

SWAP   Dn                   Vertausche Registerhälften

TAS    <ea>                 Prüfe und setze Operand
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

TRAP   #Konst               Falle

TRAPV                       Falle bei Überlauf

TST.x  <ea>                 Prüfe Operanden
S: Dn,(An),(An)+,-(An),d(An),d(An,Xi),Abs.W,Abs.L

UNLK   An                   Binden rückgängig machen
```

Anhang D
Zeichencode-Tabellen

Die RawKey-Codes (Tastatur-Scancodes)

Der ASCII-Zeichensatz

D.1 Die RawKey-Codes (Tastatur-Scancodes)

ESC $45	F1 $50	F2 $52	F3 $52	F4 $53	F5 $54	F6 $55	F7 $56	F8 $57	F9 $58	F10 $59

' $00	1 ! $01	2 @ $02	3 # $03	4 $ $04	5 % $05	6 ^ $06	7 & $07	8 * $08	9 ($09	0) $0A	- _ $0B	= + $0C	\ $0D	BACKS $41

TAB $42	Q q $10	W w $11	E e $12	R r $13	T t $14	Y y $15	U u $16	I i $17	O o $18	P p $19	[{ $1A	] } $1B	RETURN

CTRL $63	CAPSL $62	A a $20	S s $21	D d $22	F f $23	G g $24	H h $25	J j $26	K k $27	L l $28	; : $29	" ' $2A	$2B	$44

SHIFT $60	| $30	Z z $31	X x $32	C c $33	V v $34	B b $35	N n $36	M m $37	, < $38	. > $39	/ ? $3A	SHIFT $61

ALT $64	A $66	SPACE $40	A $67	ALT $65

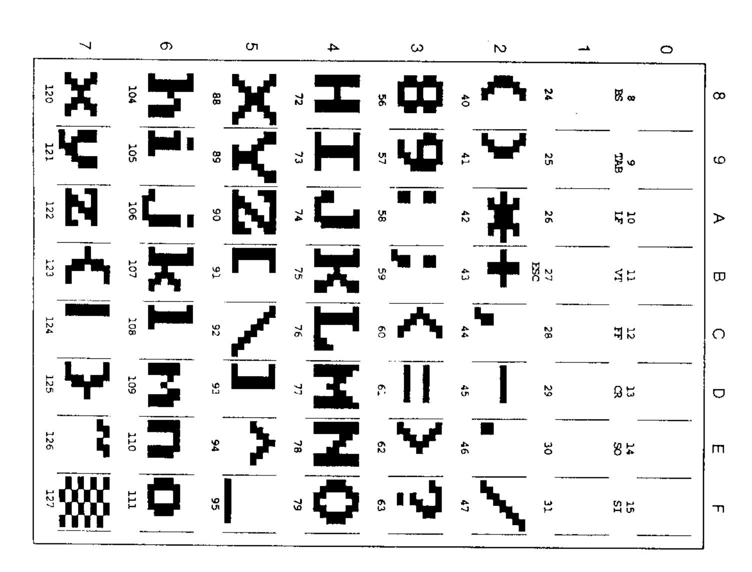
DEL $46	HELP $5F

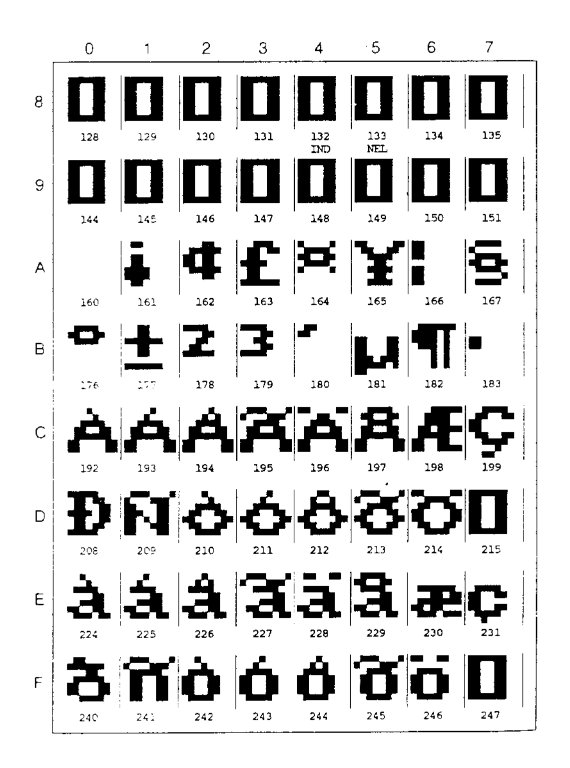
	UP $4C	
LEFT $4F	DOWN $4D	RIGTH $4E

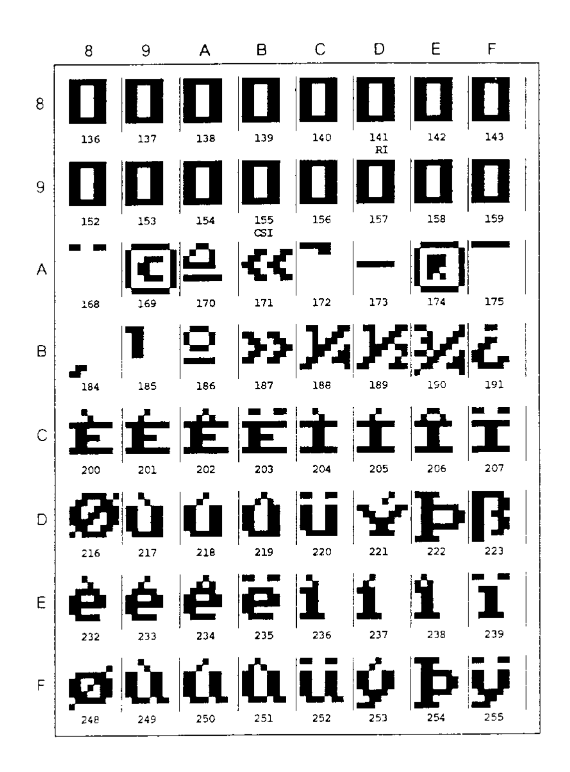
{ [$1A	}] $1B	/ $3A	* $88
7 $3D	8 $3E	9 $3F	- $4A
4 $2D	5 $2E	6 $2F	+ $0C
1 $1D	2 $1E	3 $1F	EN TER
0 $0F		DEL $3C	$43

D.2 Der ASCII-Zeichensatz






0 1 2 3 4 5 6 7
8
128 129 130 131 132 IND 133 NEL 134 135
9
144 145 146 147 148 149 150 151
A
160 161 162 163 164 165 166 167
B
176 177 178 179 180 181 182 183
C
192 193 194 195 196 197 198 199
D
208 209 210 211 212 213 214 215
E
224 225 226 227 228 229 230 231
F
240 241 242 243 244 245 246 247


8 9 A B C D E F
8
136 137 138 139 140 141 RI 142 143
9
152 153 154 155 CSI 156 157 158 159
A
168 169 170 171 172 173 174 175
B
184 185 186 187 188 189 190 191
C
200 201 202 203 204 205 206 207
D
216 217 218 219 220 221 222 223
E
232 233 234 235 236 237 238 239
F
248 249 250 251 252 253 254 255

Anhang E
Steuersequenzen

ANSI-Steuersequenzen

Drucker-Steuersequenzen

E.1 ANSI-Steuersequenzen

ANSI-Sequenz	Bedeutung
$9b,["n"],$40	n Leerzeichen ab Cursorposition einfügen
$9b,["n"],$41	Cursor n Zeilen nach oben setzen
$9b,["n"],$42	Cursor n Zeilen nach unten setzen
$9b,["n"],$43	Cursor n Zeilen nach rechts setzen
$9b,["n"],$44	Cursor n Zeilen nach links setzen
$9b,["n"],$45	Cursor hoch
$9b,["n"],$46	Cursor runter
$9b,["z"],[$3b,"s"],$48	Cursor in z (Zeile)/s (Spalte) setzen. Dabei kann die Angabe für die Spalte (s) vernachlässigt werden.
$9b,$4a	Fenster ab Cursorposition löschen
$9b,$4b	Zeile ab Cursorpositin löschen
$9b,$4c	Zeile einfügen
$9b,$4d	Zeile löschen
$9b,["n"],$50	n Zeilen löschen
$9b,["n"],$53	n Zeilen nach oben schieben
$9b,["n"],$54	n Zeilen nach unten schieben
$9b,$32,$30,$68	Linefeed in Return+Linefeed konvertieren
$9b,$32,$30,$6c	Linefeed nur zu Linefeed konvertieren
$9b,$6e	Aktuelle Cursorposition senden. Dabei erhält man eine Zeichenkette folgenden Aufbaus: $9b,"z",$3b,"s",$52 (z/s Zeile/Spalte)
$9b,"S;V;H",$6d	Stil, Vordergrundfarbe und Hintergrundfarbe setzen. s (Stil) 0 = normal 1 = fett 3 = schräg 4 = unterstrichen 7 = invertieren v (Vordergr.) 30-37 Farben 0-7 h (Hintergr.) 40-47 Farben 0-7
$9b,("h"),$74	Setzt die maximale Anzahl der Zeilen auf h fest.
$9b,("b"),$75	Setzt die maximale Zeilenlänge auf b fest.
$9b,("a"),$78	Setzt den Abstand (a) vom linken Rand des Fensters, ab dem die Ausgabe beginnen soll.
$9b,("a"),$79	Setzt den Abstand (a) vom oberen Rand des Fensters, ab dem die Ausgabe beginnen soll.
$9b,$30,$20,$70	Cursor unsichtbar machen
$9b,$20,$70	Cursor sichtbar machen
$9b,$71	Fensterausmaße senden. Dabei bekommt man eine Zeichenkette folgenden Formats wieder. $9b,$31,$3b,$31,$3b,"z",$3b,"s",$73 (z/s Zeile/Spalte)

Wenn man die in eckigen Klammern gesetzten Werte wegläßt, so wird als Parameter 1 übergeben.

Wenn man die in runden Klammern gesetzten Werte wegläßt, so wird der Standardwert wieder eingestellt..

E.2 Drucker-Steuersequenzen

Kommando	Wert	ESC-Sequenz	Bedeutung
aRIS	0	ESCc	Drucker-Reset ausführen
aRIN	1	ESC#1	Drucker initialisieren
aIND	2	ESCD	Zeilenvorschub
aNEL	3	ESCE	Wagenrücklauf + Zeilenvorschub
aRI	4	ESCM	Zeile nach oben
aSGR0	5	ESC[0m	Standard-Zeichensatz
aSGR3	6	ESC[3m	Kursivschrift ein
aSGR23	7	ESC[23m	Kursivschrift aus
aSGR4	8	ESC[4m	Unterstrichen ein
aSGR24	9	ESC[24m	Unterstrichen aus
aSGR1	10	ESC[1m	Fettdruck ein
aSGR22	11	ESC[11m	Fettdruck aus
aSFC	12	ESC[3nm	Vordergrundfarbe n (0-9) einstellen
aSBC	13	ESC[4nm	Hintergrundfarbe n (0-9) einstellen
aSHORP0	14	ESC[0w	Normale Druckbreite
aSHORP2	15	ESC[2w	Elite-Druckdichte ein
aSHORP1	16	ESC[1w	Elite-Druckdichte aus
aSHORP4	17	ESC[4w	Schmalschrift ein
aSHORP3	18	ESC[3w	Schmalschrift aus
aSHORP6	19	ESC[6w	Breitschrift ein
aSHORT5	20	ESC[5w	Breitschrift aus
aDEN6	21	ESC[6"z	Schattendruck ein
aDEN5	22	ESC[5"z	Schattendruck aus
aDEN4	23	ESC[4"z	Doppeldruck ein
aDEN3	24	ESC[3"z	Doppeldruck aus
aDEN2	25	ESC[2"z	NLQ ein
aDEN1	26	ESC[1"z	NLQ aus
aSUS2	27	ESC[2v	Hochstellen ein
aSUS1	28	ESC[1v	Hochstellen aus
aSUS4	29	ESC[4v	Tiefstellen ein
aSUS3	30	ESC[3v	Tiefstellen aus
aSUS0	31	ESC[0v	Normale Schriftstellung
aPLU	32	ESCL	Teillinie aufwärts
aPLD	33	ESCK	Teillinie abwärts
aFNT0	34	ESC(B	Zeichensatz USA
aFNT1	35	ESC(R	Zeichensatz Frankreich
aFNT2	36	ESC(K	Zeichensatz Deutschland
aFNT3	37	ESC(A	Zeichensatz England
aFNT4	38	ESC(E	Zeichensatz Dänemark 1
aFNT5	39	ESC(H	Zeichensatz Schweden
aFNT6	40	ESC(Y	Zeichensatz Italien

aFNT7	41	ESC(Z	Zeichensatz Spanien
aFNT8	42	ESC(J	Zeichensatz Japan
aFNT9	43	ESC(6	Zeichensatz Norwegen
aFNT10	44	ESC(C	Zeichensatz Dänemark 2
aPROP2	45	ESC[2p	Proportional ein
aPROP1	46	ESC[1p	Proportional aus
aPROP0	47	ESC[0p	Proportional-Einstellung löschen
aTSS	48	ESC[n E	Proportional-Offset n einstellen
aJFY5	49	ESC[5 F	Linksausrichtung
aJFY7	50	ESC[7 F	Rechtsausrichtung
aJFY6	51	ESC[6 F	Zentrierte Ausrichtung
aJFY0	52	ESC[0 F	Ausrichtung aus
aJFY2	53	ESC[2 F	Wortabstand (Zentrierung)
aJFY3	54	ESC[3 F	Buchstabenabstand (Ausrichtung)
aVERP0	55	ESC[0z	Zeilenabstand 1/8"
aVERP1	56	ESC[1z	Zeilenabstand 1/6"
aSLPP	57	ESC[nt	Seitenlänge n einstellen
aPERF	58	ESC[nq	Überspringe n (n>0) Zeilen
aPERF0	59	ESC[0q	Überspringen aus
aLMS	60	ESC#9	Linker Rand gesetzt
aRMS	61	ESC#0	Rechter Rand gesetzt
aTMS	62	ESC#8	Oberer Rand gesetzt
aBMS	63	ESC#2	Unterer Rand gesetzt
aSTBM	64	ESC[Pn;mr	Setze Ränder auf n oben, m unten
aSLRM	65	ESC[Pn;ms	Setze Ränder auf n links, m rechts
aCAM	66	ESC#3	Ränder löschen
aHTS	67	ESCH	Horizontal-Tabulator setzen
aVTS	68	ESCJ	Vertikal-Tabulator setzen
aTBC0	69	ESC[0g	Horizontal-Tabulator löschen
aTBC3	70	ESC[3g	Alle Horizontal-Tabulatoren löschen
aTBC1	71	ESC[1g	Vertikal-Tabulator löschen
aTBC4	72	ESC[4g	Alle Vertikal-Tabulatoren löschen
aTBCALL	73	ESC#4	Alle V- und H-Tabulatoren löschen
aTBSALL	74	ESC#5	Standard-Tabulatoren setzen
aEXTEND	75	ESC[Pn"x	Erweitertes Kommando n ausführen

Anhang F
System-Fehlermeldungen

DOS-Fehlermeldungen

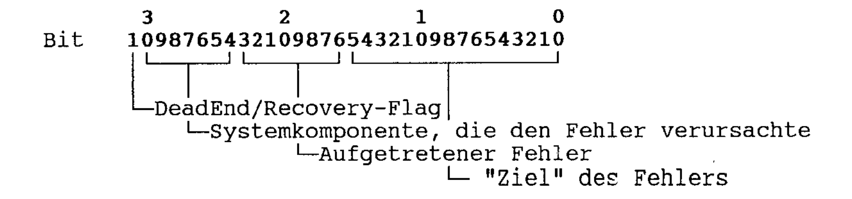
Guru-Meditation-Fehlermeldungen

F.1 Die DOS-Fehlermeldungen

DOS-Fehlermeldung	Wert	Bedeutung
NO_FREE_STORE	103	Zu wenig Speicherplatz
TASK_TABLE_FULL	105	Zu viele CLI-Tasks eingerichtet
LINE_TOO_LONG	120	CLI-Eingabezeile zu lang
FILE_NOT_OBJECT	121	Datei ist nicht ausführbar
INVALID_RESIDENT_LIBRARY	122	Residente Lib beim Laden fehlerhaft
OBJECT_IN_USE	202	Gewünschtes Objekt wird schon benutzt
OBJECT_EXISTS	203	Objekt schon vorhanden
OBJECT_NOT_FOUND	205	Objekt nicht gefunden
ACTION_NOT_KNOWN	209	Filesystem erhielt unbekanntes Packet
INVALID_COMPONENT_NAME	210	CLI-Eingabe enthält ungültige Zeichen
INVALID_LOCK	211	Ungültiges Lock verwendet
OBJECT_WRONG_TYPE	212	File als Dir angesprochen oder umgekehrt
DISK_NOT_VALIDATED	213	Diskette nicht für gültig erklärt
DISK_WRITE_PROTECTED	214	Diskette ist schreibgeschützt
RENAME_ACROSS_DEVICES	215	Rename auf anderen Datenträger versucht
DIRECTORY_NOT_EMPTY	216	Löschversuch auf nicht-leeres Verzeichnis
DEVICE_NOT_MOUNTED	218	Gerät nicht angemeldet
SEEK_ERROR	219	Seek-Aufruf außerhalb der Dateigrenzen
COMMENT_TOO_BIG	220	Dateikommentar zu lang
DISK_FULL	221	Diskette voll
DELETE_PROTECTED	222	Datei ist löschgeschützt
WRITE_PROTECTED	223	Datei ist schreibgeschützt
READ_PROTECTED	224	Datei ist lesegeschützt
NOT_A_DOS_DISK	225	Keine DOS-Diskette im Laufwerk
NO_DISK	226	Gar keine Diskette im Laufwerk
NO_MORE_ENTRIES	232	ExNext: Keine weiteren Verz.einträge

F.2 Die Guru-Meditation-Fehlermeldungen

Die Guru-Fehlermeldungen bestehen aus zwei Langworten. Das erste gibt den Fehler an, das zweite die Task-Struktur-Adresse des abstürzenden Tasks. Die Fehlernummer ist folgendermaßen aufgebaut:



F.2.1 Die allgemeinen Fehlercodes

Es gibt zwei Arten von Fehlermeldungen: Die allgemeinen und die speziellen. Auf die allgemeine Form bezieht sich die obige Grafik.

Das DeadEnd/Recovery-Flag (höchstes Bit) gibt an, ob nach der Guru-Meldung ein Reset ausgelöst wird (DeadEnd - totes Ende für alle Programme) oder das laufende Programm weitergeht (Recovery - "erholbarer" Guru).

Für die auslösende Systemkomponente gelten folgende Zuordnungen:

Systemkomponente	Wert	Bedeutung
AN_CPUTrap	$00000000	CPU-Exception
AN_ExecLib	$01000000	Exec-Library
AM_GraphicsLib	$02000000	Graphics-Library
AN_LayersLib	$03000000	Layers-Library
AN_Intuition	$04000000	Intuition-Library
AN_MathLib	$05000000	Math...-Library
AN_CListLib	$06000000	CList-Library
AN_DOSLib	$07000000	DOS-Library
AN_RAMLib	$08000000	RAMLib-Library
AN_IconLib	$09000000	Icon-Library
AN_ExpansionLib	$0a000000	Expansion-Library
AN_AudioDev	$10000000	Audio-Device
AN_ConsoleDev	$11000000	Console-Device
AN_GamePortDev	$12000000	Gameport-Device
AN_KeyboardDev	$13000000	Keyboard-Device
AN_TrackDiskDev	$14000000	Trackdisk-Device
AN_TimerDev	$15000000	Timer-Device
AN_CIARsrc	$20000000	CIA-Resource
AN_DiskRsrc	$21000000	Disk-Resource
AN_MiscRsrc	$22000000	Misc-Resource
AN_BootStrap	$30000000	Resident-Routine für Bootvorgang
AN_Workbench	$31000000	Workbench-Programm
AN_DiskCopy	$32000000	Diskcopy-Programm

Folgende Nummern gehören zu den aufgetretenen Fehlern:

Fehler	Wert	Bedeutung
AG_NoMemory	$00010000	Nicht genug Speicherplatz
AG_MakeLib	$00020000	Fehler bei Library-Erstellung
AG_OpenLib	$00030000	Fehler beim Öffnen einer Library
AG_OpenDev	$00040000	Fehler beim Öffnen eines Devices
AG_OpenRes	$00050000	Fehler beim Öffnen einer Resource
AG_IOError	$00060000	Fehler bei Bearbeitung eines IORequest
AG_NoSignal	$00070000	Kein Signal empfangen

Die Systemkomponenten, auf die sich der Fehler bezieht, bekommen folgende Nummern:

Fehlerobjekt	Wert	Bedeutung
AO_ExecLib	$00008001	Exec-Library
AO_GraphicsLib	$00008002	Graphics-Library
AO_LayersLib	$00008003	Layers-Library
AO_Intuition	$00008004	Intuition-Library
AO_MathLib	$00008005	Math...-Library
AO_CListLib	$00008006	CList-Library
AO_DOSLib	$00008007	DOS-Library
AO_RAMLib	$00008008	RAMLib-Library
AO_IconLib	$00008009	Icon-Library
AO_ExpansionLib	$0000800a	Expansion-Library
AO_AudioDev	$00008010	Audio-Device
AO_ConsoleDev	$00008011	Console-Device
AO_GamePortDev	$00008012	Gameport-Device
AO_KeyboardDev	$00008013	Keyboard-Device
AO_TrackDiskDev	$00008014	Trackdisk-Device
AO_TimerDev	$00008015	Timer-Device
AO_CIARsrc	$00008020	CIA-Resource
AO_DiskRsrc	$00008021	Disk-Resource
AO_MiscRsrc	$00008022	Misc-Resource
AO_BootStrap	$00008030	Resident-Routine für Bootvorgang
AO_Workbench	$00008031	Workbench-Programm

Die auf diese Weise ermittelten Werte müssen für die endgültige Guru-Nummer OR-verknüpft werden. Beispiel: Der DeadEnd-Fehler "DOS-Library konnte Trackdisk-Device nicht öffnen" wird kodiert als

$80000000 (DeadEnd) OR $07000000 (Quelle: DOS-Library) OR $00040000 (Fehler beim Device-Öffnen) OR $00008014 (Objekt: Trackdisk-Device)

Das ergibt die Fehlernummer $87048014.

F.2.2 Die speziellen Fehlercodes

Neben den allgemeinen Fehlercodes, die man sich aus den eben besprochenen drei Codebereichen zusammensetzen kann, gibt es für die einzelnen Systemkomponenten noch spezielle Fehlercodes, die nun aufgelistet werden sollen.

Prozessor-Exceptions

Fehler	Wert	Bedeutung
Bus Error	$00000002	Busfehler
Address Error	$00000003	Adreßfehler
Illegal Instruction	$00000004	Illegale Instruktion
Divion By Zero	$00000005	Division durch 0
CHK Instruction	$00000006	CHK-Befehl

TRAPV Instruction	$00000007	TRAPV-Befehl
Privilege Violation	$00000008	Privileg-Verletzung
Trace	$00000009	Trace-Modus
Line A Emulation	$0000000a	Befehl mit %1010-Bitfolge am Anfang
Line F Emulation	$0000000b	Befehl mit %1111-Bitfolge am Anfang

Exec-Library

Fehler	Wert	Bedeutung
AN_ExcptVect	$81000001	Checksumme der Prozessor-Exception falsch
AN_BaseChkSum	$81000002	Checksumme der ExecBase falsch
AN_LibChkSum	$81000003	Checksumme der Library falsch
AN_LibMem	$81000004	Kein Speicherplatz für Library-Erstellung
AN_MemCorrupt	$81000005	Memory-List zerstört
AN_IntrMem	$81000006	Kein Speicherplatz für Interrupt-Server
AN_InitAPtr	$81000007	Fehler bei Ausführung von InitStruct
AN_SemCorrupt	$81000008	Semaphore in fehlerhaftem Zustand
AN_FreeTwice	$81000009	Speicherbereich zweimal freigegeben
AN_BogusExcpt	$8100000a	Illegale Exception durchgeführt

Graphics-Library

Fehler	Wert	Bedeutung
AN_GfxNoMem	$82010000	Kein Speicherplatz für Graphics
AN_LongFrame	$82010006	Kein Speicher für Longframe-Copperlist
AN_ShortFrame	$82010007	Kein Speicher für Shortframe-Copperlist
AN_TextTmpRas	$02010009	Kein Speicher für Text-TmpRas
AN_BltBitMap	$8201000a	Kein Speicher für BltBitMap-Routine
AN_RegionMemory	$8201000b	Kein Speicher für Region
AN_MakeVPort	$82010030	Kein Speicher für MakeVPort-Routine
AN_GfxNoLCM	$82011234	Notfall-Speicher nicht verfügbar

Layers-Library

Fehler	Wert	Bedeutung
AN_LayersNoMem	$83010000	; Kein Speicherplatz für Layers

Intuition-Library

Fehler	Wert	Bedeutung
AN_GadgetType	$84000001	Unbekannter Gadget-Typ
AN_BadGadget	$04000001	Recovery-Form von AN_GadgetType
AN_CreatePort	$84010002	Kein Speicher für Message-Port
AN_ItemAlloc	$84010003	Kein Speicher für Item-Plane
AN_SubAlloc	$84010004	Kein Speicher für Sub
AN_PlaneAlloc	$84010005	Kein Speicher für Plane
AN_ItemBoxTop	$84000006	Oberkante der Item-Box < Nullkoordinaten
AN_OpenScreen	$84010007	Kein Speicher für neuen Screen

AN_OpenScrnRast	$84010008	Kein Speicher für neuen Screen-Raster
AN_SysScrnType	$84000009	Unbekannter Typ des System-Screens
AN_AddSWGadget	$8401000a	Kein Speicher für System-Window-Gadgets
AN_OpenWindow	$8401000b	Kein Speicher für neues Window
AN_BadState	$8400000c	"Bad State" beim Start von Intuition
AN_BadMessage	$8400000d	"Bad Message" von IDCMP empfangen
AN_WeirdEcho	$8400000e	Unbekannte Meldung
AN_NoConsole	$8400000f	Console-Device ließ sich nicht öffnen

DOS-Library

Fehler	Wert	Bedeutung
AN_StartMem	$07010001	Kein Speicherplatz in Startphase
AN_EndTask	$07000002	Fehler beim Beenden eines Tasks
AN_QPktFail	$07000003	QPacket-Fehler
AN_AsyncPkt	$07000004	Unerwartetes Packet empfangen
AN_FreeVec	$07000005	Fehler bei FreeVec-Funktion
AN_DiskBlkSeq	$07000006	Fehler in der Diskblock-Sequenz
AN_BitMap	$07000007	Bitmap fehlerhaft
AN_KeyFree	$07000008	Diskblock schon freigegeben
AN_BadChkSum	$07000009	Falsche Checksumme
AN_DiskError	$0700000a	Disk-Fehler
AN_KeyRange	$0700000b	Blocknummer nicht im erlaubten Bereich
AN_BadOverlay	$0700000c	Ungültiger Overlay

RAMLib-Library

Fehler	Wert	Bedeutung
AN_BadSegList	$08000001	Overlays in Libraries nicht erlaubt

Expansion-Library

Fehler	Wert	Bedeutung
AN_BadExpansionFree	$0a000001	Fehlerhafte Expansion-Freigabe

Trackdisk-Device

Fehler	Wert	Bedeutung
AN_TDCalibSeek	$14000001	Such-Fehler beim Kalibrieren
AN_TDDelay	$14000002	Fehler beim Warten auf Timer-Signal

Timer-Device

Fehler	Wert	Bedeutung
AN_TMBadReq	$15000001	Fehlerhafte Anfrage
AN_TMBadSupply	$15000002	Stromversorgung unterstützt Ticks nicht

Disk-Resource

Fehler	Wert	Bedeutung
AN_DRHasDisk	$21000001	GetUnit hat bereits Diskzugriff
AN_DRIntNoAct	$21000002	Interrupt-Fehler: Keine active Unit

Bootstrap

Fehler	Wert	Bedeutung
AN_BootError	$30000001	Fehler beim Bootvorgang

Anhang G
Datenstrukturen und Flags

G.1 Diskfont-Strukturen und -Flags

TextAttr

```
00      dc.l    *ta_Name               ; Zeiger auf Font-Name
04      dc.w     ta_YSize              ; Vertikale Größe des Fonts
06      dc.b     ta_Style              ; Stil des Fonts
07      dc.b     ta_Flags              ; Art des Fonts
08               ta_SIZEOF
```

Font-Flag	Wert	Bedeutung
FPF_ROMFONT	1	Font ist im ROM
FPF_DISKFONT	2	Font wurde von Disk geladen
FPF_REVPATH	4	Ausrichtung von rechts nach links
FPF_TALLDOT	8	Hires-Noninterlaced-Font
FPF_WIDEDOT	16	Lores-Interlaces-Font
FPF_PROPORTIONAL	32	Proportional-Font
FPF_DESIGNED	64	Font ist gezeichnet, nicht berechnet
FPF_REMOVED	128	Font wurde per Lib-Routine entfernt

TextFont

```
00      ds.b     tf_Message,20         ; Eine Exec-Message (siehe unten)
20      dc.w     tf_YSize              ; Y-Größe des Fonts
22      dc.b     tf_Style              ; Schreibstil
23      dc.b     tf_Flags              ; Font-Eigenschaften
24      dc.w     tf_XSize              ; Standard-Fontbreite
26      dc.w     tf_Baseline           ; Grundlinie
28      dc.w     tf_BoldSmear
30      dc.w     tf_Accessors          ; Anzahl der Zugriffe
32      dc.b     tf_LoChar             ; Erstes ASCII-Zeichen im Font
33      dc.b     tf_HiChar             ; Letztes ASCII-Zeichen im Font
34      dc.l    *tf_CharData           ; Zeiger auf die Zeichen-Daten
38      dc.w     tf_Modulo
40      dc.l    *tf_CharLoc
44      dc.l    *tf_CharSpace          ; Zeiger auf Proportional-Daten
48      dc.l    *tf_CharKern           ; Zeiger auf Kerning-Daten
52               tf_SIZEOF
```

AvailFontsHeader

```
00      dc.w    afh_NumEntries         ; Anzahl der folgenden AF-Strukturen
02      ds.b    afh_AF,10              ; Die erste AvailFonts-Struktur
12      ds.b    afh_AF,10              ; Die zweite AvailFonts-Struktur
..      ....    .........              ; usw.
```

AvailFonts

```
00      dc.w     af_Type              ; AFF_MEMORY oder AFF_DISK
02      dc.l    *af_Name,0            ;┐
06      dc.w     af_YSize             ;│ TextAttr-
08      dc.b     af_Style             ;│   Struktur
09      dc.b     af_Flags             ;┘
10               af_SIZEOF
```

Font-Typ	Wert	Bedeutung
AFF_MEMORY	1	Suche Fonts im ROM
AFF_DISK	2	Suche Fonts auf der Diskette

FontContentsHeader

```
000     dc.w    fch_FileID            ; $0f00
002     dc.w    fch_NumEntries        ; Anzahl der folgenden fc-Strukturen
004     ds.b    fch_FH,260            ; Die erste FC-Struktur
264     ds.b    fch_FH,260            ; Die zweite FC-Struktur
...     ....    ..........            ; usw.
```

FontContents

```
000     ds.b    fc_FileName,256       ; Name der Fontdatei (0-terminiert)
256     dc.w    fc_YSize              ; Y-Größe
258     dc.b    fc_Style              ; Schreibstil
259     dc.b    fc_Flags              ; Font-Eigenschaften
260             fc_SIZEOF
```

DiskFontHeader

```
000     ds.b    dfh_DF,14             ; Ein Exec-Node (siehe unten)
014     dc.w    dfh_FileID            ; $0f80
016     dc.w    dfh_Revision          ; Revisionsnummer des Fonts
018     dc.l    dfh_Segment           ; Segment-Adresse nach Laden
022     ds.b    dfh_Name,256          ; Name der Fontdatei
278     ds.b    dfh_TF,52             ; Eine TextFont-Struktur
330             dfh_SIZEOF
```

G.2 DOS-Strukturen und -Flags

FileInfoBlock (FIB)

```
000     dc.l    fib_DiskKey           ; Nummer des Verwaltungsblocks
004     dc.l    fib_DirEntryType      ; > 0 bei Verz., sonst Datei
008     ds.b    fib_FileName,108      ; Dateiname, max. 30 Zeichen
116     dc.l    fib_Protection        ; Schutzstatus
120     dc.l    fib_EntryType
```

```
124     dc.l    fib_Size                ; Dateilänge in Bytes
128     dc.l    fib_NumBlocks
132     ds.b    fib_DateStamp,12        ; Zeitpunkt der letzten Änderung
144     ds.b    fib_Comment,116         ; Dateikommentar
260             fib_SIZEOF
```

InfoData

```
00      dc.l    id_NumSoftErrors        ; Anzahl "weicher" Fehler
04      dc.l    id_UnitNumber           ; Nummer des Laufwerks
08      dc.l    id_DiskState            ; Disketten-Zustand, siehe unten
12      dc.l    id_NumBlocks            ; Anzahl Datenblöcke
16      dc.l    id_NumBlocksUsed        ; Anzahl belegter Blöcke
20      dc.l    id_BytesPerBlock        ; Anzahl Bytes in einem Block
24      dc.l    id_DiskType             ; Disketten-Typ, siehe unten
28      dc.l    id_VolumeNode
32      dc.l    id_InUse                ; 1 = Disk wird benutzt
36              id_SIZEOF
```

Diskstatus	Wert	Bedeutung
ID_WRITE_PROTECTED	80	Schreibgeschützt
ID_VALIDATING	81	Wird gerade auf Gültigkeit überprüft
ID_VALIDATED	82	Ist gültig

Disk-Typ	Wert	Bedeutung
ID_NO_DISK_PRESENT	-1	Keine Disk im Laufwerk
ID_UNREADABLE_DISK	$42414400	Disk nicht lesbar
ID_NOT_REALLY_DOS	$4E444F53	Keine DOS-Disk
ID_DOS_DISK	$444F5300	Disk o.k.
ID_FFS_DISK	$444F5301	FastFileSystem-Disk
ID_KICKSTART_DISK	$4B49434B	A1000-Kickstart-Disk

Datei-Zugriff

Öffnungs-Modus	Wert	Bedeutung
MODE_OLDFILE	1005	Öffnet bestehende Datei
MODE_READWRITE	1004	Öffnet neue oder bestehende Datei
MODE_NEWFILE	1006	Erstellt neue Datei

Offset-Angaben für Seek

Offset-Typ	Wert	Bedeutung
OFFSET_BEGINNING	-1	Relativ zum Dateianfang
OFFSET_CURRENT	0	Relativ zur derzeitigen Position
OFFSET_END	1	Relativ zum Ende

Lock-Typen

Lock-Typ	Wert	Bedeutung
EXCLUSIVE_LOCK	-1	Exklusiv-Recht des aufrufenden Programms
ACCESS_WRITE	-1	Siehe EXCLUSIVE_LOCK
SHARED_LOCK	-2	Kein Exklusiv-Recht
ACCESS_READ	-2	Siehe SHARED_LOCK

Protection-Attribute

Schutzattribut	Wert	Bedeutung
FIBF_DELETE	1	Löschschutz
FIBF_EXECUTE	2	Ausführungsschutz
FIBF_WRITE	4	Schreib-(Veränderungs-)schutz
FIBF_READ	8	Leseschutz
FIBF_ARCHIVE	16	Datei ist archiviert
FIBF_PURE	32	Datei ist resident ladbar
FIBF_SCRIPT	64	Datei ist eine Batch-Datei
FIBF_HIDE	128	Datei soll nicht angezeigt werden

DOS-Library-Basis

```
00      ds.b     dl_lib,34            ; Library-Struktur
34      dc.l    *dl_Root              ; Zeiger auf Root-Node
38      dc.l    *dl_GV                ; Zeiger auf 'Global Vector'
42      dc.l     dl_A2                ; DOS-interne Register-
46      dc.l     dl_A5                ; zwischenspeicher
50      dc.l     dl_A6
54               dl_SIZEOF
```

RootNode

```
00      dc.l    *rn_TaskArray         ; BCPL-Zeiger auf CLI-Task-Tabelle
04      dc.l    *rn_ConsoleSegment    ; BCPL-Zeiger auf Konsolen-Handler
08      ds.b     rn_Time,12           ; Systemzeit im DateStamp-Format
20      dc.l    *rn_RestartSeg        ; BCPL-Zeiger auf Disk-Validator
24      dc.l    *rn_Info              ; BCPL-Zeiger auf DosInfo-Struktur
28      dc.l    *rn_FileHandlerSegment; BCPL-Zeiger auf File-Handler
32               rn_SIZEOF
```

DosInfo

```
00      dc.l    *di_McName            ; Derzeit nicht benutzt
04      dc.l    *di_DevInfo           ; BCPL-Zeiger auf Device-List
08      dc.l    *di_Devices           ; Derzeit nicht benutzt
12      dc.l    *di_Handlers          ; Derzeit nicht benutzt
16      dc.l    *di_NetHand           ; Derzeit nicht benutzt
20               di_SIZEOF
```

DeviceNode

```
00    dc.l   *dn_Next            ; BCPL-Zeiger auf nächste Struktur
04    dc.l    dn_Type            ; Struktur-Typ (bei Devices 0)
08    dc.l   *dn_Task            ; Zeiger auf Device-Task
12    dc.l   *dn_Lock            ; Für Devices nicht benutzt
16    dc.l   *dn_Handler         ; Handler-Dateiname als BCPL-String
20    dc.l    dn_StackSize       ; Stack-Größe für neue Tasks
24    dc.l    dn_Priority        ; Priorität für neue Tasks
28    dc.l   *dn_Startup         ; BCPL-Zeiger auf FileSysStartupMsg
32    dc.l   *dn_SegList         ; BCPL-Zeiger auf Handler-Segment
36    dc.l   *dn_GlobalVec       ; Zeiger auf globalen Vektor
40    dc.l   *dn_Name            ; Name des Devices als BCPL-String
44            dn_SIZEOF
```

DevList (VolumeNode)

```
00    dc.l   *dl_Next            ; BCPL-Zeiger auf nächste Struktur
04    dc.l    dl_Type            ; Struktur-Typ (für Disks 2)
08    dc.l   *dl_Task            ; BCPL-Zeiger auf Handler-Task
12    dc.l   *dl_Lock            ; Für Disks nicht benutzt
16    ds.b    dl_VolumeDate,12   ; Erstellungsdatum (DateStamp!)
28    dc.l   *dl_LockList        ; BCPL-Zeiger auf Disk-Locks
32    dc.l    dl_DiskType        ; Diskettentyp (z.B. $444F5300)
36    dc.l    dl_unused          ; Nicht belegt
40    dc.l   *dl_Name            ; Diskname als BCPL-String
44            dl_SIZEOF
```

FileSysStartupMsg

```
00    dc.l    fssm_Unit          ; Exec-Unit-Nummer (für OpenDevice)
04    dc.l   *fssm_Device        ; Devicename als BCPL-String
08    dc.l   *fssm_Environ       ; BCPL-Zeiger auf Environment-Tab.
12    dc.l    fssm_Flags         ; Flags für OpenDevice
14            fssm_SIZEOF
```

Disk-Environment

```
00    dc.l   de_TableSize        ; Anzahl Langworte in der Tabelle
04    dc.l   de_SizeBlock        ; Größe eines Blocks in Langworten
08    dc.l   de_SecOrg           ; Nicht benutzt; muß 0 sein
12    dc.l   de_NumHeads         ; Anzahl der Schreib/Lese-Köpfe
16    dc.l   de_SecsPerBlk       ; Nicht benutzt, muß 1 sein
20    dc.l   de_BlksPerTrack     ; Anzahl Blöcke auf einer Spur
24    dc.l   de_ReservedBlks     ; Anzahl Sektoren im Bootblock
28    dc.l   de_Prefac           ; Nicht benutzt, Muß 0 sein
32    dc.l   de_Interleave       ; Gewöhnlich 0
36    dc.l   de_LowCyl           ; Nummer des ersten Zylinders
40    dc.l   de_UpperCyl         ; Nummer des letzten Zylinders
44    dc.l   de_NumBuffers       ; Anzahl der Speicherpuffer
48    dc.l   de_MemBufType       ; Speichertyp für Puffer
52           de_SIZEOF
```

Process

```
000   dc.l   *tc_Succ          ;┐
004   dc.l   *tc_Pred          ;│
008   dc.b    tc_Type          ;├── Node-Struktur
009   dc.b    tc_Pri           ;│
010   dc.l   *tc_Name          ;┘
014   dc.b    tc_Flags         ;
015   dc.b    tc_State         ;
016   dc.b    tc_IDNestCnt     ;
017   dc.b    tc_TDNestCnt     ;
018   dc.l    tc_SigAlloc      ;
022   dc.l    tc_SigWait       ;
026   dc.l    tc_SigRecvd      ; ├── Task-Struktur
030   dc.l    tc_Except        ;       (pr_Task)
034   dc.w    tc_TrapAlloc     ;
036   dc.w    tc_TrapAble      ;
038   dc.l    tc_ExceptData    ;
042   dc.l    tc_ExceptCode    ;
046   dc.l    tc_TrapData      ;
050   dc.l    tc_TrapCode      ;
054   dc.l    tc_SPReg         ;
058   dc.l    tc_SPLower       ;
062   dc.l    tc_SPUpper       ;
066   dc.l   *tc_Switch        ;
070   dc.l   *tc_Launch        ;
074   dcb.b   tc_MemEntry,14   ;
088   dc.l    tc_UserData      ;┘

092   dc.l   *mp_Succ          ;┐
096   dc.l   *mp_Pred          ;│
100   dc.b    mp_Type          ;│
101   dc.b    mp_Pri           ;│
102   dc.l   *mp_Name          ;├── MessagePort-Struktur
106   dc.b    mp_Flags         ;│       (pr_MsgPort)
107   dc.b    mp_SigBit        ;│
108   dc.l   *mpSigTask        ;│
112   dcb.b   mpMsgList,14     ;┘

126   dc.w    pr_Pad              ;    Füllwort
128   dc.l   *pr_SegList          ; BP Zeiger auf Segmentliste
132   dc.l    pr_StackSize        ;    Größe des Stacks
136   dc.l    pr_GlobVec          ;    globaler Vektor
140   dc.l    pr_TaskNum          ;    Nummer des CLI-Tasks
144   dc.l    pr_StackBase        ; BP obere Grenze des Stacks
148   dc.l    pr_Result2          ;    zweiter Rückgabewert
152   dc.l   *pr_CurrentDir       ; BP Zeiger auf Lock
156   dc.l   *pr_CIS              ; BP Zeiger auf Ausgabe-FileH
160   dc.l   *pr_COS              ; BP Zeiger auf Eingabe-FileH
164   dc.l   *pr_ConsoleTask      ;    Zeiger auf Console-Hd
168   dc.l   *pr_FileSystemTask   ;    Zeiger auf File-System-Task
172   dc.l   *pr_CLI              ; BP Zeiger auf CLI-Struktur
176   dc.l   *pr_RrturnAddr       ;    Zeiger auf End-Routine
180   dc.l   *pr_PktWait          ;    eigene "Wait"-Routine
184   dc.l   *pr_WindowPtr        ;    Zeiger auf Fenster
188           pr_SIZEOF
```

Name	Bit #	Bedeutung
CTRL_C	12	Ctrl-C gedrückt
CTRL_D	13	Ctrl-D gedrückt
CTRL_E	14	Ctrl-E gedrückt
CTRL_F	15	Ctrl-F gedrückt

CLI

```
00   dc.l    cli_Result2          ;    Fehlernummer
04   dc.l   *cli_SetName          ; BP aktuelles Direktory
08   dc.l   *cli_CommandDir       ; BP Zeiger auf Command-Lock
12   dc.l    cli_ReturnCode       ;    FAILAT-Wert
16   dc.l   *cli_CommandName      ; BP Name des Befehls
20   dc.l    cli_FailLevel        ;    Fehlergrenze
24   dc.l   *cli_Prompt           ; BP Zeiger auf Prompt-Text
28   dc.l   *cli_StandardInput    ; BP Standard-Eingabekanal
32   dc.l   *cli_CurrentInput     ; BP umgeleitete Eingabe
36   dc.l   *cli_CommandFile      ; BP Name der Batchdatei
40   dc.l    cli_Interactive      ;    Interactive-Flag
44   dc.l    cli_Background       ;    Background-Flag
48   dc.l   *cli_CurrentOutput    ; BP umgeleitete Ausgabe
52   dc.l    cli_DefaultStack     ;    Default-Stackgröße
56   dc.l   *cli_StandardOutPut   ; BP Standard-Ausgabekanal
60   dc.l   *cli_Module           ; BP Zeiger auf 1. Segment
64           cli_SIZEOF
```

Name	Wert	Bedeutung
RETURN_OK	0	kein Fehler
RETURN_WARN	5	Warnung
RETURN_ERROR	10	Fehler aufgetreten
RETURN_FAILT	20	totaler Fehler bzw. mehrere Fehler

DosPacket

```
00   dc.l   *dp_Link              ; Zeiger auf die Message-Struktur
04   dc.l   *dp_Port              ; Zeiger auf ReplyPort
08   dc.l    dp_Type/dp_Action    ; Kommando
12   dc.l    dp_Res1/dp_Status    ;┐Rückgabewerte
16   dc.l    dp_Res2/dp_Status2   ;┘
20   dc.l    dp_Arg1/dp_BufAddr   ;┐
24   dc.l    dp_Arg2              ;│
28   dc.l    dp_Arg3              ;│ Argumente
32   dc.l    dp_Arg4              ;│
36   dc.l    dp_Arg5              ;│
40   dc.l    dp_Arg6              ;│
44   dc.l    dp_Arg7              ;┘
48           dp_SIZEOF
```

WBStartup

```
00      ds.b     sm_Message,20        ; Message-Struktur
20      dc.l    *sm_Preocess          ; Zeiger auf Process-Descriptor
24      dc.l    *sm_Segment           ; Zeiger auf Segment-Descriptor
28      dc.l     sm_NumArgs           ; Anzahl der Elemente der ArgList
32      dc.l    *sm_ToolWindow        ; Zeiger auf Fenster-Descriptor
36      dc.l    *sm_ArgList           ; Zeiger auf Argumentenliste
40               sm_SIZEOF
```

G.3 Exec-Strukturen und -Flags

G.4 Graphics-Strukturen und -Flags

RastPort

```
000     dc.l    *rp_Layer             ; Zeiger auf den Layer des Rastports
004     dc.l    *rp_BitMap            ; Zeiger auf zugehörige BitMap
008     dc.l    *rp_AreaPtrn          ; Zeiger auf Füllmuster
012     dc.l    *rp_TmpRas            ; Zeiger auf Zwischenspeicher
016     dc.l    *rp_AreaInfo          ; Zeiger auf Area-Struktur
020     dc.l    *rp_GelsInfo          ; Zeiger auf GelsInfo-Struktur
024     dc.b     rp_Mask
025     dc.b     rp_FgPen             ; Vordergrundfarbe
026     dc.b     rp_BgPen             ; Hintergrundfarbe
027     dc.b     rp_AOLPen            ; Begrenzungsfarbe beim Füllen
028     dc.b     rp_DrawMode          ; Zeichenmodus
029     dc.b     rp_AreaPtSz          ; Anzahl Worte im Füllmuster
030     dc.b     rp_Dummy             ; Nicht benutzt
031     dc.b     rp_linpatcnt
032     dc.w     rp_Flags             ; System-Flags
034     dc.w     rp_LinePtrn          ; Linienmuster
036     dc.w     rp_cp_x              ; x-Position des Zeichencursors
038     dc.w     rp_cp_y              ; y-Position des Zeichencursors
040     ds.b     rp_minterms,8        ; Blittermaske (Grafik-Hardware)
048     dc.w     rp_PenWidth          ; Breite des Zeichenstiftes
050     dc.w     rp_PenHeight         ; Höhe des Zeichenstiftes
052     dc.l    *rp_Font              ; Zeiger auf TextFont-Struktur
056     dc.b     rp_AlgoStyle         ; Font-Stil
057     dc.b     rp_TxFlags           ; Font-Flags
058     dc.w     rp_TxHeight          ; Höhe des Fonts
060     dc.w     rp_TxWidth           ; Breite des Fonts
062     dc.w     rp_TxBaseline        ; Position der Font-Grundlinie
064     dc.w     rp_TxSpacing         ; Leer-Abstand der Fontzeichen
066     dc.l    *rp_RP_User           ; Zeiger auf Reply-Port
070     ds.b     rp_reserved,30       ; Reserviert für Erweiterungen
100              rp_SIZEOF
```

TmpRas (Temporal Rastport)

```
00      dc.l    *tr_RasPtr          ; Zeiger auf den Speicherbereich
04      dc.l     tr_Size            ; Größe dieses Bereichs
08               tr_SIZEOF
```

AreaInfo

```
00      dc.l    *ai_VctrTbl         ; Beginn der Vektortabelle
04      dc.l    *ai_VctrPtr         ; Nächster Vektoreintrag
08      dc.l    *ai_FlagTbl         ; Beginn der Flagtabelle
12      dc.l    *ai_FlagPtr         ; Nächster Flageintrag
16      dc.w     ai_Count           ; Derzeitige Vektoreintragsnummer
18      dc.w     ai_MaxCount        ; Maximalzahl Vektoreinträge
20      dc.w     ai_FirstX          ; x-Koord. des ersten Punktes
22      dc.w     ai_FirstY          ; y-Koord. des ersten Punktes
24               ai_SIZEOF
```

Drawmode

Zeichenmodus	Wert	Bedeutung
JAM1	0	Normal
JAM2	1	Gelöschte Punkte in Hintergrundfarbe
COMPLEMENT	2	Vorhandene Grafik NOT-verknüpfen
INVERSVID	4	Zu zeichnende Grafik invertieren

Schriftstil

Schriftstil	Wert	Bedeutung
FSF_NORMAL	0	Kein besonderer Stil
FSF_UNDERLINED	1	Unterstrichen
FSF_BOLD	2	Fettdruck
FSF_ITALIC	4	Kursiv-(Schräg-)Druck
FSF_EXTENDED	8	Doppelte Breite (bei normalen Fonts nicht möglich)

ViewMode

View-Mode	Wert	Bedeutung
GENLOCK_VIDEO	$0002	Bindet eine externe Signalquelle ein
EXTRA_HALFBRITE	$0080	64-Farben Modus
DUALPF	$0400	Dual-Playfield
Hold-And-Modify	$0800	HAM-Modus (4096 Farben)
VP_HIDE	$2000	Kein Bild
SPRITES	$4000	View mit Hardware-Sprites
HIRES	$0004	Verdoppelt Auflösung in x-Richtung
LACE	$8000	Verdoppelt Auflösung in y-Richtung

View

```
00    dc.l   *v_ViewPort        ; Zeiger auf den ersten ViewPort
04    dc.l   *v_LOFCprList      ; Zeiger auf Longframe-Copperlist
08    dc.l   *v_SHFCprList      ; Zeiger auf Shortframe-Copperlist
12    dc.w    v_DyOffset        ; y-Startkoordinate des View
14    dc.w    v_DxOffset        ; x-Startkoordinate des View
16    dc.w    v_Modes           ; Bildschirmauflösung etc.
18            v_SIZEOF
```

ViewPort

```
00    dc.l   *vp_Next           ; Zeiger auf den nächsten ViewPort
04    dc.l   *vp_ColorMap       ; Zeiger auf ColorMap-Struktur
08    dc.l   *vp_DspIns         ; Zeiger auf Display-Copperliste
12    dc.l   *vp_SprIns         ; Zeiger auf Sprite-Copperliste
16    dc.l   *vp_ClrIns         ; Zeiger auf Color-Copperliste
20    dc.l   *vp_UCopIns        ; Zeiger auf User-Copperliste
24    dc.w    vp_DWidth         ; Breite des ViewPorts
26    dc.w    vp_DHeight        ; Höhe des ViewPorts
28    dc.w    vp_DxOffset       ; x-Startkoordinate
30    dc.w    vp_DyOffset       ; y-Startkoordinate
32    dc.w    vp_Modes          ; Bildschirmauflösung etc.
34    dc.w    vp_reserved       ; Reserviert für Erweiterungen
36    dc.l   *vp_RasInfo        ; Zeiger auf RasInfo-Struktur
40            vp_SIZEOF
```

ColorMap

```
00    dc.b    cm_Flags          ; Interne Flags
01    dc.b    cm_Type           ; Betriebssystemsversion
02    dc.w    cm_Count          ; Anzahl der Farbeinträge
04    dc.l   *cm_ColorTable     ; Zeiger auf Farbtabelle
08            cm_SIZEOF
```

BitMap

```
00    dc.w    bm_BytesPerRow    ; Anzahl Bytes in einer Grafikzeile
02    dc.w    bm_Rows           ; Anzahl Zeilen in der Grafik
04    dc.b    bm_Flags          ; System-Flags
05    dc.b    bm_Depth          ; Tiefe, Anzahl Bitplanes
06    dc.b    bm_Pad            ; Reserviert für Erweiterungen
08    ds.l   *bm_Planes,8       ; 8 Langwort-Planezeiger
40            bm_SIZEOF
```

RasInfo

```
00    dc.l   *ri_Next           ; Zeiger auf nächste RasInfo
04    dc.l   *ri_BitMap         ; Zeiger auf Bitmap
08    dc.w    ri_RxOffset       ; x-Koordinate des Rasters
10    dc.w    ri_RyOffset       ; y-Koordinate des Rasters
12            ri_SIZEOF
```

SimpleSprite

```
00    dc.l   *ss_posctldata      ; Zeiger auf Grafik-Daten
04    dc.w    ss_height          ; Höhe
06    dc.w    ss_x               ; x-Position
08    dc.w    ss_y               ; y-Position
10    dc.w    ss_num             ; Sprite-Nummer
12            ss_SIZEOF
```

GelsInfo

```
00    dc.b    gi_sprRsrvd        ; Zu benutzende Hardware-Sprites
01    dc.b    gi_Flags
02    dc.l   *gi_gelHead         ; Erstes Gel
06    dc.l   *gi_gelTail         ; Letztes Gel
10    dc.l   *gi_nextLine        ; System-Zeiger
14    dc.l   *gi_lastColor       ; System-Zeiger
18    dc.l   *gi_collHandler     ; Zeiger auf Sprungtabelle für Koll.
22    dc.w    gi_leftmost        ; Linker Rand
24    dc.w    gi_rightmost       ; Rechter Rand
26    dc.w    gi_topmost         ; Oberer Rand
28    dc.w    gi_bottommost      ; Unterer Rand
30    dc.l   *gi_firstBlissObj
34    dc.l   *gi_lastBlissObj
38            gi_SIZEOF
```

VSprite (für VSprites)

```
00    dc.l   *vs_NextVSprite     ; Zeiger auf den nächsten VSprite
04    dc.l   *vs_PrevVSprite     ; Zeiger auf den letzten VSprite
08    dc.l   *vs_DrawPath        ; ---
12    dc.l   *vs_ClearPath       ; ---
16    dc.w    vs_Oldy            ; ---
18    dc.w    vs_Oldx            ; ---
20    dc.w    vs_VSFlags         ; VSprite-Flags
22    dc.w    vs_Y               ; y-Koordinate
24    dc.w    vs_X               ; x-Koordinate
26    dc.w    vs_Height          ; Höhe in Pixeln
28    dc.w    vs_Width           ; Breite in Words (immer 1)
30    dc.w    vs_Depth           ; Tiefe (immer 2)
32    dc.w    vs_MeMask          ; Kollisionsmaske 1
34    dc.w    vs_HitMask         ; Kollisionsmaske 2
36    dc.l   *vs_ImageData       ; Zeiger auf Grafikdaten
40    dc.l   *vs_BorderLine      ; Zeiger auf BorderLine-Wort
44    dc.l   *vs_CollMask        ; Zeiger auf CollMask-Plane
48    dc.l   *vs_SprColors       ; Zeiger auf Wortfeld für Farben
52    dc.l   *vs_Bob             ; ---
56    dc.b    vs_PlanePick       ; ---
57    dc.b    vs_PlaneOnOff      ; ---
58            vs_SIZEOF
```

VSprite-Flag	Wert	Bedeutung
VSF_VSPRITE	$001	Gesetzt für VSprite, gelöscht für Bob

VSF_MUSTDRAW	$008	VSprite muß gezeichnet werden
VSF_GELGONE	$400	VSprite ragt über Begrenzung hinaus
VSF_OVERFLOW	$800	Zu viele VSprites in einer Zeile

VSprite (für Bobs)

00	dc.l	*vs_NextVSprite	; Zeiger auf den nächsten Bob
04	dc.l	*vs_PrevVSprite	; Zeiger auf den letzten Bob
08	dc.l	*vs_DrawPath	; Reihenfolge beim Zeichnen
12	dc.l	*vs_ClearPath	; Reihenfolge beim Löschen
16	dc.w	vs_Oldy	; Vorige y-Koordinate
18	dc.w	vs_Oldx	; Vorige x-Koordinate
20	dc.w	vs_VSFlags	; VSprite-Flags
22	dc.w	vs_Y	; y-Koordinate
24	dc.w	vs_X	; x-Koordinate
26	dc.w	vs_Height	; Höhe in Pixeln
28	dc.w	vs_Width	; Breite in Words
30	dc.w	vs_Depth	; Tiefe
32	dc.w	vs_MeMask	; Kollisionsmaske 1
34	dc.w	vs_HitMask	; Kollisionsmaske 2
36	dc.l	*vs_ImageData	; Zeiger auf Grafikdaten
40	dc.l	*vs_BorderLine	; Zeiger auf BorderLine-Wort
44	dc.l	*vs_CollMask	; Zeiger auf CollMask-Plane
48	dc.l	*vs_SprColors	; ---
52	dc.l	*vs_Bob	; Zeiger auf Bob-Struktur
56	dc.b	vs_PlanePick	; Maske für benutze Planes
57	dc.b	vs_PlaneOnOff	; Plane-Maske zum Ein/Ausschalten
58		vs_SIZEOF	

Bob

00	dc.w	bob_BobFlags	; Bob-Flags
02	dc.l	*bob_SaveBuffer	; Speicher zur Bitmap-Sicherung
06	dc.l	*bob_ImageShadow	; Zeigt auf die CollMask
10	dc.l	*bob_Before	; Für Zeichenreihenfolge
14	dc.l	*bob_After	; Für Zeichenreihenfolge
18	dc.l	*bob_VSprite	; Zeiger zurück zu VSprite
22	dc.l	*bob_BobComp	; Zeiger auf AnimComp-Struktur
26	dc.l	*bob_DBuffer	; Zeiger für double-buffering
30		bob_SIZEOF	

Bob-Flag	Wert	Bedeutung
VSF_VSPRITE	$001	Gesetzt für VSprite, gelöscht für Bob
VSF_SAVEBACK	$002	Bitmap unter Bob sichern
VSF_OVERLAY	$004	Bob-Pixel in Farbe 0 durchsichtig
VSF_BACKSAVED	$100	Bitmap unter Bob wurde gesichert
VSF_BOBUPDATE	$200	System-intern
VSF_GELGONE	$400	VSprite ragt über Begrenzung hinaus
BF_SAVEBOB	$0001	Hintergrund nicht restaurieren
BF_BOBISCOMP	$0002	Bob gehört zu einer Animation
BF_BWAITING	$0100	Intern
BF_BDRAWN	$0200	Intern

BF_BOBSAWAY	$0400	Bob beim nächsten Zeichnen entfernen
BF_BOBNIX	$0800	Bob wurde entfernt
BF_SAVEPRESERVE	$1000	Intern
BF_OUTSTEP	$2000	Intern

Gel-Kollisionen mit Rahmen

Hit-Flag	Wert	Bedeutung
TopHit	1	Obere Begrenzung überschritten
BottomHit	2	Untere Begrenzung überschritten
LeftHit	4	Linke Begrenzung überschritten
RightHit	8	Rechte Begrenzung überschritten

Graphics-Library-Basis

```
000   ds.b    gb_LibNode,34          ; Library-Struktur
034   dc.l   *gb_ActiView            ; Zeiger auf aktuellen View
038   dc.l   *gb_copinit             ; Zeiger auf Copper-Startup-Liste
042   dc.l   *gb_cia                 ; intern
046   dc.l   *gb_blitter             ; intern
050   dc.l   *gb_LOFlist             ; Zeiger auf LOF-Copperliste
054   dc.l   *gb_SHFlist             ; Zeiger auf SHF-Copperliste
058   dc.l   *gb_blthd               ; intern
062   dc.l   *gb_blttl               ; intern
066   dc.l   *gb_bsblthd             ; intern
070   dc.l   *gb_bsblttl             ; intern
074   ds.b    gb_vbsrv,22            ; Interrupt-Server für Vert. Blank
096   ds.b    gb_timsrv,22           ; Interrupt-Server für Timer
118   ds.b    gb_bltsrv,22           ; Interrupt-Server für Blitter
140   ds.b    gb_TextFonts,14        ; Listenkopf der Textfont-Liste
154   dc.l   *gb_DefaultFont         ; Zeiger auf Standard-Font
158   dc.w    gb_Modes               ; intern
160   dc.b    gb_VBlank              ; intern
161   dc.b    gb_Debug               ; intern
162   dc.w    gb_BeamSync            ; intern
164   dc.w    gb_system_bplcon0      ; intern
166   dc.b    gb_SpriteReserved      ; intern
167   dc.b    gb_bytereserved        ; Füllbyte
168   dc.w    gb_Flags               ; Library-interne Flags
170   dc.w    gb_BlitLock            ; intern
172   dc.w    gb_BlitNest            ; intern
174   ds.b    gb_BlitWaitQ,14        ; Interner Listenkopf
188   dc.l   *gb_BlitOwner           ; Zeiger auf Blitter-Besitzertask
192   ds.b    gb_TOF_WaitQ,14        ; Interner Listenkopf
206   dc.w    gb_DisplayFlags        ; Darstellungsmodus
208   dc.l   *gb_SimpleSprites       ; Zeiger auf SimpleSprites
212   dc.w    gb_MaxDisplayRow       ; Maximalzahl Bildschirmzeilen
214   dc.w    gb_MaxDisplayColumn    ; Maximalzahl Bildschirmspalten
216   dc.w    gb_NormalDisplayRow    ; Standardwert Bildschirmzeilen
218   dc.w    gb_NormalDisplayCol    ; Standardwert Bildschirmspalten
220   dc.w    gb_NormalDPMX          ; intern
222   dc.w    gb_NormalDPMY          ; intern
224   dc.l   *gb_LastChanceMemory    ; Zeiger auf "Notfall-Speicher"
```

```
228     dc.l    *gb_LCMptr           ; intern
232     dc.w     gb_MicrosPerLine    ; intern
234     ds.b     gb_reserved,8       ; Für zukünftige Erweiterungen
242              gb_SIZEOF
```

G.5 Intuition-Strukturen und -Flags

NewWindow

```
00      dc.w     nw_LeftEdge         ; linke Ecke
02      dc.w     nw_TopEdge          ; obere Ecke
04      dc.w     nw_Width            ; Breite
06      dc.w     nw_Height           ; Höhe
08      dc.b     nw_DetailPen1       ; Vordergrundfarbe
09      dc.b     nw_BlockPen         ; Hintergrundfarbe
10      dc.l     nw_IDCMPFlags       ; IDCMP-Flags
14      dc.l     nw_Flag             ; Flags
18      dc.l    *nw_FirstGadget      ; Zeiger auf erstes Gadget
22      dc.l    *nw_CheckMark        ; Grafik für Menühaken
26      dc.l    *nw_Title            ; Name des Festers
30      dc.l    *nw_Screen           ; Zeiger auf Screen
34      dc.l    *nw_BitMap           ; Zeiger auf eigene BitMap
38      dc.w     nw_MinWidth         ; X-Minimum des Fensters
40      dc.w     nw_MinHeight        ; Y-Minimum des Fensters
42      dc.w     nw_MaxWidth         ; X-Maximum des Fensters
44      dc.w     nw_MaxHeight        ; Y-Maximum des Fensters
46      dc.w     nw_Type             ; Art des Screens, auf dem das
                                     ; Window erscheint
48               nw_SIZEOF
```

IDCMP-Flag	Wert	Bedeutung
SIZEVERIFY	$00000001	Größe des Fensters soll verändert werden
NEWSIZE	$00000002	Größe des Fensters wurde verändert
REFRESHWINDOW	$00000004	Fenster wurde überlagert
ACTIVEWINDOW	$00040000	Fenster wurde aktiviert
INACTIVEWINDOW	$00080000	Fenster wurde deaktiviert
GADGETDOWN	$00000020	GADGIMMADIATE-Gadget wurde gewählt
GADGETUP	$00000040	RELVERIFY-Gadget wurde angewählt
CLOSEWINDOW	$00000200	Close-Gadget wurde angewählt
REQSET	$00000080	Erster Requester wurde geöffnet
REQCLEAR	$00001000	Letzter Requester wurde geschlossen
REQVERIFY	$00000800	Requester soll geöffnet werden
MENUPICK	$00000100	Menüpunkt wurde gewählt
MENUVERIFY	$00002000	Menü soll gezeigt werden
MOUSEBUTTONS	$00000008	Eine Maustaste wurde gedrückt
MOUSEMOVE	$00000010	Maus wurde bewegt
DELTAMOVE	$00100000	Mausbewegung relativ
INTUITICKS	$00400000	Jede zehntel Sekunde ein Nachricht
NEWPREFS	$00004000	Preferences wurden geändert
DISKINSERTED	$00008000	Diskette wurde eingelegt
DISKREMOVED	$00010000	Diskette wurde entnommen

RAWKEY	$00000400	Tastatureingabe mit RAW-Codes
VANILLAKEY	$00200000	Eingabe mit bearbeiteten KeyCodes
WBENCHMESSAGE	$00020000	Nachricht von der WBench
LONELYMESSAGE	$80000000	Keine IDCMP-Nachricht

Windowflag	Wert	Bedeutung
SIZEBRIGHT	$00000010	Size-Gadget im rechten Rand
SIZEBOTTOM	$00000020	Size-Gadget im unteren Rand
WINDOWSIZING	$00000001	Gadget für Größeneinstellung
WINDOWDRAG	$00000002	Fenster kann verschoben werden
WINDOWDEPTH	$00000004	DEPTH-Gadget wird eingebunden
WINDOWCLOSE	$00000008	Close-Gadget wird eingebunden
BACKDROP	$00000100	Fenster direkt auf Screen legen
GIMMEZEROZERO	$00000400	Getrennte Verwaltung Inhalt & Rand
BORDERLESS	$00000800	Fenster ohne Ränder darstellen
ACTIVATE	$00001000	Fenster wird beim Öffnen aktiv
REPORTMOUSE	$00000200	Mauskoordinaten werden gemeldet
RMBTRAP	$00010000	RightMouseButtonTRAP (kein Menü)
NOCAREREFRESH	$00020000	Keine Meldung wenn Fenster beschädigt
SIMPLE_REFRESH	$00000040	Fensterinhalt wird nicht erneuert
SMART_REFRESH	$00000000	verdeckte Bereiche sichern
SUPER_BITMAP	$00000080	Restauration aus eigener BitMap möglich

Window

```
000   dc.l   *wd_NextWindow       ; Zeiger auf nächstes Window
004   dc.w    wd_LeftEdge         ; linke Ecke
006   dc.w    wd_TopEdge          ; obere Ecke
008   dc.w    wd_Width            ; Breite
010   dc.w    wd_Height           ; Höhe
012   dc.w    wd_MouseY           ; Y-Mauskoordinate
014   dc.w    wd_MouseX           ; X-Mauskoordinate
016   dc.w    wd_MinWidth         ; minimale Breite
018   dc.w    wd_MinHeight        ; minimale Höhe
020   dc.w    wd_MaxWidth         ; maximale Breite
022   dc.w    wd_MaxHeight        ; maximale Höhe
024   dc.l    wd_Flags            ; Window-Flags
028   dc.l   *wd_MenuStrip        ; Zeiger auf Menü-Struktur
032   dc.b   *wd_Title            ; Zeiger auf Titelzeile
036   dc.l   *wd_FirstRequester   ; Zeiger auf ersten Requester
040   dc.l   *wd_DMRequest        ; Zeiger auf Double-Menu-Req.
044   dc.w    wd_ReqCount         ; Zähler der Requester
046   dc.l   *wd_WScreen          ; Zeiger auf Screen
050   dc.l   *wd_RPort            ; Zeiger auf RastPort
054   dc.b    wd_BorderLeft       ;
055   dc.b    wd_BorderTop        ;
056   dc.b    wd_BorderRight      ;
057   dc.b    wd_BorderBottom     ;
058   dc.l   *wd_BorderRPort      ; Zeiger auf Border RastPort
062   dc.l   *wd_FirstGadget      ; Zeiger auf erstes Gadget
066   dc.l   *wd_Parent           ; vorhergehendes Fenster
070   dc.l   *wd_Descendant       ; nachfolgendes Fenster
074   dc.w   *wd_Pointer          ; Zeiger auf Mausdaten
078   dc.b    wd_PtrHeight        ; Höhe des Mauszeigers
```

```
079    dc.b    wd_PtrWidth          ; Breite des Mauszeigers
080    dc.b    wd_XOffset           ; X-Koordinate des HOTSPOT
081    dc.b    wd_YOffset           ; Y-Koordinate des HOTSPOT
082    dc.l    wd_IDCMPFlag         ; IDCMP-Flags
086    dc.l   *wd_UserPort          ; Zeiger auf UserPort
090    dc.l   *wd_WindowPort        ; Zeiger auf WindowPort
094    dc.l   *wd_MessageKey        ; Zeiger auf MessageKey
098    dc.b    wd_DetailPen         ; Farbe für Vordergrund
099    dc.b    wd_BlockPen          ; Farbe für Hintergrund
100    dc.l   *wd_CheckMark         ; Zeiger auf Grafikdaten
104    dc.b   *wd_ScreenTitle       ; Zeiger auf Screentitel
108    dc.w    wd_GZZMouseX         ; Mauskoordinaten für GZZ
110    dc.w    wd_GZZMouseY         ;
112    dc.w    wd_GZZWidth          ;
114    dc.w    wd_GZZHeight         ;
116    dc.b   *wd_ExtData           ; Zeiger auf externe Daten
120    dc.b   *wd_UserData          ; Zeiger auf eigene Daten
124    dc.l   *wd_WLayer            ; Zeiger auf Window-Layer
128    dc.l   *wd_IFont             ; Zeiger auf TextFontStruktur
132            wd_SIZEOF
```

<u>Intui-Message</u>

```
00     dc.l   *ln_Succ              ;┐      ┐
04     dc.l   *ln_Pred              ;│      │
08     dc.b    ln_Type              ;│ Node │
09     dc.b    ln_Pri               ;│      │ Message
10     dc.l   *ln_Name              ;┘      │
14     dc.l   *mn_ReplyPort         ;       │
18     dc.w    mn_Length            ;       ┘
20     dc.l    im_Class             ; IDCMP-Flag der Nachricht
24     dc.w    im_Code              ; Nachrichten-abhängige Daten
26     dc.w    im_Qualifier         ;
28     dc.l    im_IAddress          ; Zeiger auf den Auslöser
32     dc.w    im_MouseX            ; Mauskoordinate (X)
34     dc.w    im_MouseY            ; Mauskoordinate (Y)
36     dc.l    im_Seconds           ; Sekunden
40     dc.l    im_Micros            ; Mikros
44     dc.l   *im_IDCMPWindow       ; Zeiger auf Fenster
48     dc.l   *im_SpecialLink       ; Systemspezifisch
52             im_SIZEOF
```

<u>NewScreen</u>

```
00     dc.w    ns_LeftEdge          ; X-Koordinate des Screens
02     dc.w    ns_TopEdge           ; Y-Koordinate des Screens
04     dc.w    ns_Width             ; Breite
06     dc.w    ns_Height            ; Höhe
08     dc.w    ns_Depth             ; Anzahl BitPlanes
10     dc.b    ns_DetailPen         ; Farben für den Vordergrund
11     dc.b    ns_BlockPen          ; und den Hintergrund
12     dc.w    ns_ViewModes         ; Auflösung
14     dc.w    ns_Type              ; Screen-Typ
16     dc.l   *ns_Font              ; TextAttr-Struktur
```

```
20      dc.l    *ns_DefaultTitle        ; Zeiger auf Titelzeile
24      dc.l    *ns_Gadgets             ; Custom-Gadgets
28      dc.l    *ns_CustomBitMap        ; eigene BitMap-Struktur
32               ns_SIZEOF
```

Screentyp	Wert	Bedeutung
WBENCHSCREEN	$0001	Screen ist der WBench-Screen
CUSTOMSCREEN	$000F	Screen ist Custom-Screen
CUSTOMBITMAP	$0040	Keine BitMap erstellen
SCREENBEHIND	$0080	Screen im Hintergurnd öffnen
SCREENQUIET	$0100	System-Gadgets/Titelzeile abschalten
SHOWTITLE	$0010	Intuition-intern
BEEPING	$0020	Intuition-intern

View-Mode	Wert	Bedeutung
GENLOCK_VIDEO	$0002	Bindet eine externe Signalquelle ein
EXTRA_HALFBRITE	$0080	64-Farben Modus
DUALPF	$0400	Dual-Playfield
Hold-And-Modify	$0800	HAM-Modus (4096 Farben)
VP_HIDE	$2000	Kein Bild
SPRITES	$4000	Screen mit Hardware-Sprites
HIRES	$0004	Verdoppelt Auflösung in X-Richtung
LACE	$8000	Verdoppelt Auflösung in Y-Richtung

Screen

```
000     dc.l    *sc_NextScreen          ; Zeiger auf nächsten Screen
004     dc.l    *sc_FirstWindow         ; Zeiger auf erstes Window
008     dc.w     sc_LeftEdge            ; linke Ecke
010     dc.w     sc_TopEdge             ; rechte Ecke
012     dc.w     sc_Width               ; Breite
014     dc.w     sc_Height              ; Höhe
016     dc.w     sc_MouseY              ; Y-Mauskoordinaten
018     dc.w     sc_MouseX              ; X-Mauskoordinaten
020     dc.w     sc_Flags               ; Screen-Flags
022     dc.l    *sc_Title               ; Zeiger auf Titelstring
026     dc.l    *sc_DefaultTitle        ; Default-Titelstring
030     dc.b     sc_BarHeight           ; Höhe der Titelleiste
031     dc.b     sc_BarVBorder          ; vertikaler Rand
032     dc.b     sc_BarHBorder          ; horizontaler Rand
033     dc.b     sc_MenuVBorder         ; vertikaler Rand (Menü)
034     dc.b     sc_MenuHBorder         ; horizontaler Rand (Menü)
035     dc.b     sc_WBorTop             ; Breite, Window-Rand oben
036     dc.b     sc_WBorLeft            ; Breite, Window-Rand links
037     dc.b     sc_WBorRight           ; Breite, Window-Rand rechts
038     dc.b     sc_WBorBottom          ; Breite, Window-Rand unten
039     dc.b     sc_KludgeFill00        ; Füllbyte
040     dc.l    *sc_Font                ; TextAttr-Struktur
044     ds.b     sc_ViewPort,40         ; ViewPort-Struktur
084     ds.b     sc_RastPort,100        ; RastPort-Struktur
184     ds.b     sc_BitMap,40           ; BitMap-Struktur
224     ds.b     sc_LayerInfo,92        ; LayerInfo-Struktur
316     dc.l    *sc_FirstGadget         ; Zeiger auf erstes Gadget
```

```
320     dc.b     sc_DetailPen          ; Farbtab.nr. für Vordergrund
321     dc.b     sc_BlockPen           ; Farbtab.nr. für Hintergrund
322     dc.w     sc_SaveColor0         ; BEEPING SaveMem
324     dc.l    *sc_BarLayer           ; Zeiger auf Layer
328     dc.l    *sc_ExtData            ; Zeiger auf externe Daten
332     dc.l    *sc_UserData           ; Zeiger für UserDaten
336              sc_SIZEOF
```

IntuiText

```
00      dc.b     it_FrontPen           ; Farben
01      dc.b     it_BackPen            ;
02      dc.b     it_DrawMode           ; Zeichenmodus
03      dc.b     it_KludgeFill00       ; Füllbyte
04      dc.w     it_LeftEdge           ; relative X-Koordinate
06      dc.w     it_TopEdge            ; relative Y-Koordinate
08      dc.l    *it_ITextFont          ; TextAttr-Struktur
12      dc.l    *it_IText              ; Zeiger auf Zeichenkette
16      dc.l    *it_NextIText          ; nächste IntuiText-Struktur
20               it_SIZEOF
```

Zeichenmodus	Wert	Bedeutung
JAM1	0	Normal
JAM2	1	Gelöschte Punkte in Hintergrundfarbe
COMPLEMENT	2	Vorhandene Grafik NOT-verknüpfen
INVERSVID	4	Zu zeichnende Grafik invertieren

Border

```
00      dc.w     bd_LeftEdge           ; X-Koordinate
02      dc.w     bd_TopEdge            ; Y-Koordinate
04      dc.b     bd_FrontPen           ; Vordergrund
05      dc.b     bd_BackPen            ; Hintergrund
06      dc.b     bd_DrawMode           ; Zeichenmodus
07      dc.b     bd_Count              ; Anzahl der XY-Paare
08      dc.l    *bd_XY                 ; Zeiger auf XY-Paare
12      dc.l    *bd_NextBorder         ; Zeiger auf nächsten Border
16               bd_SIZEOF
```

Image

```
00      dc.w     ig_LeftEdge           ; X-Koordinate
02      dc.w     ig_TopEdge            ; Y-Koordinate
04      dc.w     ig_Width              ; Breite des Images
06      dc.w     ig_Height             ; Höhe des Images
08      dc.w     ig_Depth              ; Tiefe (Anzahl der BitMaps)
10      dc.l    *ig_ImageData          ; Zeiger auf ImageData
14      dc.b     ig_PlanePick          ; PlanePick-Daten
15      dc.b     ig_PlaneOnOff         ; PlaneOnOff-Daten
16      dc.l    *ig_NextImage          ; Zeiger auf nächstes Image
20               ig_SIZEOF
```

Gadget

```
00      dc.l    *gg_NextGadget          ; Zeiger auf nächstes Gadget
04      dc.w     gg_LeftEdge            ; X-Position
06      dc.w     gg_TopEdge             ; Y-Position
08      dc.w     gg_Width               ; Breite des Gadgets
10      dc.w     gg_Height              ; Höhe des Gadgets
12      dc.w     gg_Flags               ; Gadget-Flags
14      dc.w     gg_Activation          ; Activation-Flags
16      dc.w     gg_GadgetType          ; Gadgettyp
18      dc.l    *gg_GadgetRender        ; "normale" Datenstruktur
22      dc.l    *gg_SelectRender        ; "aktivierte" Datenstruktur
26      dc.l    *gg_GadgetText          ; Zeiger auf IntuiText
30      dc.l     gg_MutualExclude       ; MutualExclude-Daten
34      dc.l     gg_SpecialInfo         ; SpecialInfo
38      dc.w     gg_GadgetID            ; Gadgetidentifikationsnummer
40      dc.l     gg_User                ; User-Daten
44               gg_SIZEOF
```

Gadget-Flag	Wert	Bedeutung
GADGHCOMP	$0003	Hit-Box bei Aktivierung invertieren
GADGHBOX	$0001	Hit-Box wird umrandet
GADGHIMAGE	$0002	SelectRender-Grafik wird angezeigt
GADGHNONE	$0003	keine Reaktion
GRELBOTTOM	$0008	Y-Position relativ zum Boden
GRELRIGHT	$0010	X-Position relativ zur rechten Seite
GRELWIDTH	$0020	Breite proportional zum Fenster
GRELHEIGHT	$0040	Höhe proportional zum Fenster
SELECTED	$0080	Gadget aktiviert
GADGDISABLED	$0100	Gadget nicht auswählbar
GADGIMAGE	$0004	GadgetRender und SelectRender sind Zeiger auf Image-Strukturen

Activation-Flag	Wert	Bedeutung
TOGGLESELECT	$0100	Schalter-Gadget statt Taster-Gadget
RELVERIFY	$0001	Taste wurde über Gadget losgelassen
GADGIMMEDIATE	$0002	Gadget wird direkt aktiviert
RIGHTBORDER	$0010	Gadget in rechten Rand des Fensters
LEFTBORDER	$0020	Gadget in linken Rand des Fensters
TOPBORDER	$0040	Gadget in oberen Rand des Fensters
BOTTOMBORDER	$0080	Gadget in unteren Rand des Fensters
STRINGCENTER	$0200	Zeichenkette zentrieren
STRINGRIGHT	$0400	Zeichenkette rechtsbündig
LONGINT	$0800	String-Gadget zu Integer-Gadget
ALTKEYMAP	$1000	Tastaturtabelle auf Eingabe anwenden
BOOLEXTEND	$2000	Zusatzstruktur für Boolean-Gadgets
ENDGADGET	$0004	ENDGADGET des Requesters
FOLLOWMOUSE	$0008	Mausposition wird gemeldet

Gadgettyp-Flag	Wert	Bedeutung
BOOLGADGET	$0001	Boolean-Gadget
GADGET0002	$0002	(noch nicht benutzt)
PROPGADGET	$0003	Proportional-Gadget

STRGADGET	$0004	String-Gadget
SYSGADGET	$8000	System-Gadgets
SCRGADGET	$4000	Screen-Gadget
GZZGADGET	$2000	Gadget für GZZ-Window
REQGADGET	$1000	Requester-Gadget
SIZING	$0010	Size-Gadget
WDRAGGING	$0020	Gadget für Window Verschiebung
SDRAGGING	$0030	Gadget für Screen Verschiebung
WDOWNBACK	$0060	Gadget WindowToBack
SDOWNBACK	$0070	Gadget ScreenToBack
WUPFRONT	$0040	Gadget WindowToFront
SUPFRONT	$0050	Gadget ScreenToFront
CLOSE	$0080	Close-Gadget

StringInfo

```
00      dc.l    *si_Buffer          ; Zeiger auf Text-Puffer
04      dc.l    *si_UndoBuffer      ; Zeiger auf Undo-Puffer
08      dc.w     si_BufferPos       ; Position im Puffer
10      dc.w     si_MaxChars        ; maximale Anzahl Zeichen
12      dc.w     si_DispPos         ; Pos. des ersten Zeichens
14      dc.w     si_UndoPos         ; Position im Undo-Puffer
16      dc.w     si_NumChars        ; Anzahl Zeichen im Puffer
18      dc.w     si_DispCount       ; Ausgabeposition
20      dc.w     si_CLeft           ; relative X-Position zum Win
22      dc.w     si_CTop            ; relative Y-Position zum Win
24      dc.l    *si_LayerPtr        ; Zeiger auf Layer
28      dc.l     si_LongInt         ; Zahlernwert der Eingabe
32      dc.l    *si_AltKeyMap       ; eigene Tastaturtabelle
36               si_SIZEOF
```

PropInfo

```
00      dc.w    pi_Flags            ; Flags
02      dc.w    pi_HorizPot         ; X-Position des Schalters
04      dc.w    pi_VertPot          ; Y-Position des Schalters
06      dc.w    pi_HorizBody        ; X-Größe des Schalters
08      dc.w    pi_VertBody         ; Y-Größe des Schalters
10      dc.w    pi_CWidth           ; Breite des Gadgets
12      dc.w    pi_CHeight          ; Höhe des Gadgets
14      dc.w    pi_HPotRes          ; Schrittweite in X-Richtung
16      dc.w    pi_VPotRes          ; Schrittweite in Y-Richtung
18      dc.w    pi_LeftBorder       ; Position des Rahmens (X)
20      dc.w    pi_TopBorder        ; Position des Rahmens (Y)
22              pi_SIZEOF
```

Prop-Flags	Wert	Bedeutung
FREEHORIZ	$0002	horizontale Bewegungen erlauben
FREEVERT	$0004	vertikale Bewegungen erlauben
AUTOKNOB	$0001	Schieber wird von Intuition erstellt
PROPBORDERLESS	$0008	keinen Rahmen um das Gadget zeichnen
KNOBHIT	$0100	Schieber ist betätigt worden

Menu

```
00      dc.l   *mu_NextMenu        ; Zeiger auf nächstes Menü
04      dc.w    mu_LeftEdge        ; X-Position
06      dc.w    mu_TopEdge         ; Y-Position
08      dc.w    mu_Width           ; Breite des Menüs
10      dc.w    mu_Height          ; Höhe des Menüs
12      dc.w    mu_Flags           ; Flags
14      dc.l   *mu_MenuName        ; Zeiger auf Menüname
18      dc.l   *mu_FirstItem       ; Zeiger auf ersten Menupunkt
22      dc.w    mu_JazzX           ; private Variablen Intuition
24      dc.w    mu_JazzY           ; private Variablen Intuition
26      dc.w    mu_BeatX           ; private Variablen Intuition
28      dc.w    mu_BeatY           ; private Variablen Intuition
30              mu_SIZEOF
```

Menü-Flags	Wert	Bedeutung
MENUENABLED	$0001	Menüpunkt ist nicht anwählbar
MIDDRAWN	$0100	Menüpunkt wird gerade angezeigt

MenuItem

```
00      dc.l   *mi_NextItem        ; nächste MenuItem-Struktur
04      dc.w    mi_LeftEdge        ; X-Koordinate des Punktes
06      dc.w    mi_TopEdge         ; Y-Koordinate des Punktes
08      dc.w    mi_Width           ; Breite des Menüpunktes
10      dc.w    mi_Height          ; Höhe des Menüpunktes
12      dc.w    mi_Flags           ; Flags des Menüpunktes
14      dc.l    mi_MutualExclude   ; MutualExclude-Daten
18      dc.l    mi_ItemFill        ; Grafikdaten "normal"
22      dc.l    mi_SelectFill      ; Grafikdaten "aktiviert"
26      dc.b    mi_Command         ; Tastaturcode
27      dc.b    mi_KludgeFill00    ; Füllbyte
28      dc.l   *mi_SubItem         ; Zeiger auf Untermenü
32      dc.l    mi_NextSelect      ; Nächster ausgewählter Menüpunkt
36              mi_SIZEOF
```

Menuitem-Flag	Wert	Bedeutung
CHECKIT	$0001	Menüpunkt abhaken
ITEMTEXT	$0002	IntuiText-Strukturen werden verwendet
COMMSEQ	$0004	Menü kann durch Tasten angewählt werden
MENUTOGGLE	$0008	der Zustand wird umgedreht
ITEMENABLED	$0010	Menü ist wählbar
HIGHIMGE	$0000	bei Aktivierung neues Image anzeigen
HIGHCOMP	$0040	bei Aktivierung Bereich invertieren
HIGHBOX	$0080	bei Aktivierung Bereich umranden
HIGHNONE	$00C0	bei Aktivierung passiert nichts
HIGHITEM	$2000	Menüpunkt ist gerade aktiviert
CHECKED	$0100	Menüpunkt ist gewählt
ISDRAWN	$1000	Menüpunkt wird dargestellt
MENUTOGGLED	$4000	Menüpunkt wurde schon umgedreht

Requester

```
000   dc.l   *rq_OlderRequest     ; Zeiger auf Requester
004   dc.w    rq_LeftEdge         ; X-Position
006   dc.w    rq_TopEdge          ; Y-Position
008   dc.w    rq_Width            ; Breite
010   dc.w    rq_Height           ; Höhe
012   dc.w    rq_RelLeft          ; Maus, relative X-Position
014   dc.w    rq_RelTop           ; Maus, relative Y-Position
016   dc.l   *rq_ReqGadget        ; Zeiger auf das erste Gadget
020   dc.l   *rq_ReqBorder        ; Zeiger auf einen Border
024   dc.l   *rq_ReqText          ; Zeiger auf einen Text
028   dc.w    rq_Flags            ; Flags
030   dc.b    rq_BackFill         ; Farb.Nr. des Hintergrundes
031   dc.b    rq_KludgeFill00     ; Füllbyte
032   dc.l   *rq_ReqLayer         ; Zeiger auf Layer-Struktur
036   ds.b    rq_ReqPad1,32       ; Reservierter Bereich
068   dc.l   *rq_ImageBMap        ; Zeiger auf eigenen BitMap
072   dc.l   *rq_RWindow          ; Zeiger auf das Req.-Fenster
076   ds.b    rq_ReqPad2,36       ; Reservierter Bereich
112           rq_SIZEOF
```

Requester-Flag	Wert	Bedeutung
POINTREL	$0001	Position ist relativ zur Maus
PREDRAWN	$0002	Eigene BitMap einbinden
NOISYREQ	$0004	
REQOFFWINDOW	$1000	Requester außerhalb des Fensters
REQACTIVE	$2000	Requester aktiviert
SYSREQUEST	$4000	Requester ist ein System-Requester
DEFERREFRESH	$8000	Refreshmodus wird gestoppt

Intuition-Library-Basis

```
000   dc.l   *ln_Succ             ;┐            ┐
004   dc.l   *ln_Pred             ;│            │
008   dc.b    ln_Type             ;│ Node       │
009   dc.b    ln_Pri              ;│            │
010   dc.l   *ln_Name             ;┘            │
014   dc.w    lib_Flags           ;             │
016   dc.w    lib_NegSize         ;             │ Library-Struktur
018   dc.w    lib_PosSize         ;             │
020   dc.w    lib_Version         ;             │
022   dc.w    lib_Revison         ;             │
024   dc.l   *lib_idString        ;             │
028   dc.l    lib_Sum             ;             │
032   dc.w    lib_OpenCnt         ;             ┘
034   dc.l   *ib_ViewPort         ; Erster Viewport
038   dc.l   *ib_LOFCprList       ; Haupt-Copperliste
042   dc.l   *ib_SHFCprList       ; Interlace-Zweitcopperliste
046   dc.w    ib_DyOffset         ; y-Offset des Intui-View
048   dc.w    ib_DxOffset         ; x-Offset des Intui-View
050   dc.w    ib_Modes            ; Erlaubte Viewmodes
052   dc.l   *ib_ActiveWindow     ; aktives Fenster
056   dc.l   *ib_ActiveScreen     ; aktiver Screen
```

```
060    dc.l    *ib_FirstScreen        ; erster Screen
064    dc.l     ib_Flags
068    dc.w     ib_MouseY             ; Y-Mausposition
070    dc.w     ib_MouseX             ; X-Mausposition
072    dc.l     ib_Seconds            ; Systemzeit
076    dc.l     ib_Micros             ;

       ; Der Datenteil der Intuition-Library ist zwar an dieser Stelle
       ; noch nicht beendet, doch sind die Werte, die jetzt folgen,
       ; nicht festgelegt.
```

G.6 Device-Strukturen und -Flags

G.6.1 Trackdisk-Device

<u>IOStdReq</u>

```
00     ds.b     io_Message,20         ; Zugehörige Message-Struktur
20     dc.l    *io_Device             ; Zeiger auf Device
24     dc.l    *io_Unit               ; Zeiger auf Unit
28     dc.w     io_Command            ; Trackdisk-Kommando
30     dc.b     io_Flags              ; Flags
31     dc.b     io_Error              ; Eventueller Fehler
32     dc.l     io_Actual             ; Diverse Rückgabewerte
36     dc.l     io_Length             ; Anzahl der zu übertragenden Bytes
40     dc.l    *io_Data               ; Zeiger auf Daten im Speicher
44     dc.l     io_Offset             ; Offsetwert für Device
48              io_SIZEOF
```

TDD-Kommando	Wert	Bedeutung
CMD_RESET	1	Device zurücksetzen
CMD_READ	2	Lese Daten von Diskette
CMD_WRITE	3	Schreibe Daten auf Diskette
CMD_UPDATE	4	Entleere Datenpuffer auf Disk
CMD_CLEAR	5	Lösche Datenpuffer ohne Rückschreiben
CMD_STOP	6	Device in Wartezustand bis 'Start'
CMD_START	7	Device reaktivieren nach 'Stop'
CMD_FLUSH	8	Alle IO-Kommandos abbrechen
TD_MOTOR	9	Laufwerksmotor ein- und ausschalten
TD_SEEK	10	Schreib/Lesekopf auf Spur positionieren
TD_FORMAT	11	Spuren formatieren
TD_REMOVE	12	Fehlerhafte Funktion (siehe weiter unten)!
TD_CHANGENUM	13	Anzahl Diskwechsel feststellen
TD_CHANGESTATE	14	Test, ob Disk im Laufwerk
TD_PROTSTATUS	15	Test, ob Disk schreibgeschützt
TD_RAWREAD	16	Lesen ohne anschließende Dekodierung
TD_RAWWRITE	17	Schreiben ohne vorhergehende Kodierung
TD_GETDRIVETYPE	18	Laufwerkstyp ermitteln
TD_GETNUMTRACKS	19	Maximalzahl Tracks ermitteln

TD_ADDCHANGEINT	20	Diskwechsel-Interrupt einbinden
TD_REMCHANGEINT	21	Diskwechsel-Interrupt entfernen

TDD-Flag	Wert	Bedeutung
TD_ALLOW_NON_3_5	1	Zulassung von nicht-3 1/2 Zoll-Laufw.
IOTD_INDEXSYNC	16	Indexlock-Synchronisation einschalten
IOTD_WORDSYNC	32	$4489-Wortsynchronisation einschalten

TDD-Fehler	Wert	Bedeutung
TDERR_NotSpecified	20	Allgemeiner Fehler beim Hardware-Zugriff
TDERR_NoSecHdr	21	Kein Sektorheader gefunden
TDERR_BadSecPreamble	22	Fehlerhafter Sektorvorspann
TDERR_BadSecID	23	Fehlerhafte Sektorkennmarke
TDERR_BadHdrSum	24	Sektorheader-Checksumme falsch
TDERR_BadSecSum	25	Checksumme über Blockdaten falsch
TDERR_TooFewSecs	26	Zu wenige Sektoren in einem Track
TDERR_BadSecHdr	27	Sektorheader fehlerhaft
TDERR_WriteProt	28	Schreibversuch auf schreibgeschützte Disk
TDERR_DiskChanged	29	Keine Disk im Laufwerk
TDERR_SeekError	30	Spur 0 kann nicht gefunden werden
TDERR_NoMem	31	Zu wenig Speicherplatz für Diskoperation
TDERR_BadUnitNum	32	Gewünschte Unitnummer nicht vorhanden
TDERR_BadDriveType	33	Laufwerkstyp wird vom TDD nicht unterstützt
TDERR_DriveInUse	34	Laufwerk wird schon benutzt
TDERR_PostReset	35	Device in Reset-Phase

Laufwerkstyp	Wert	Bedeutung
DRIVE3_5	1	Normales 3 1/5 Zoll-Laufwerk
DRIVE5_25	2	5 1/4 Zoll-Laufwerk

Extended Commands

Kommando	Nummer	Kommando	Nummer
ETD_READ	32770	ETD_WRITE	32771
ETD_UPDATE	32772	ETD_CLEAR	32773
ETD_MOTOR	32777	ETD_SEEK	32778
ETD_FORMAT	32779	ETD_RAWREAD	32784
ETD_RAWWRITE	32785		

IOExtTD

```
00    ds.b    iotd_IOStdReq,48     ; IOStdReq-Struktur (wie gewöhnlich)
48    dc.l    iotd_Count           ; Zähler für Diskwechsel
52    dc.l   *iotd_SecLabel        ; 16-Byte-Datenpuffer (siehe unten)
56            iotd_SIZEOF
```

TDD-Unit

```
00   ds.b  tdu_MsgPort,34     ; ┐
34   dc.b  tdu_Flags          ; │ Allgemeine
35   dc.b  tdu_Pad            ; │ Unit-Struktur
36   dc.w  tdu_OpenCnt        ; ┘
38   dc.w  tdu_Comp01Track    ; Track für erste Precompensation
40   dc.w  tdu_Comp10Track    ; Track für zweite Precompensation
42   dc.w  tdu_Comp11Track    ; Track für dritte Precompensation
44   dc.l  tdu_StepDelay      ; Verzögerungszeit beim Steppen
48   dc.l  tdu_SettleDelay    ; Verzögerungszeit nach dem Steppen
52   dc.b  tdu_RetryCnt       ; Wiederholversuche bei Fehlern
53         tdu_SIZEOF
```

G.6.2 Printer-Device

IOStdReq für Printer

```
00   ds.b   io_Message,20     ; Zugehörige Message-Struktur
20   dc.l  *io_Device         ; Zeiger auf Device
24   dc.l  *io_Unit           ; Zeiger auf Unit
28   dc.w   io_Command        ; Printer-Kommando
30   dc.b   io_Flags          ; nicht benutzt
31   dc.b   io_Error          ; Eventueller Fehler
32   dc.l   io_Actual         ; nicht benutzt
36   dc.l   io_Length         ; Anzahl der zu übertr. Zeichen
40   dc.l  *io_Data           ; Zeiger auf Daten im Speicher
44   dc.l   io_Offset         ; nicht benutzt
48          io_SIZEOF
```

Printer-Kommando	Wert	Bedeutung
CMD_WRITE	3	Normalen Text ausdrucken
PRD_RAWWRITE	9	Nicht-vorbehandelten Text ausdrucken
PRD_PRTCOMMAND	10	Standard-Druckerkommando senden
PRD_DUMPRPORT	11	Hardcopy eines Rastports ausdrucken

Printer-Fehler	Wert	Bedeutung
PDERR_CANCEL	1	Drucker nicht druckbereit
PDERR_NOTGRAPHICS	2	Drucker kann keine Grafik drucken
PDERR_INVERTHAM	3	HAM-Grafik kann nicht invertiert werden
PDERR_BADDIMENSION	4	Ungültige Druck-Grafikgröße
PDERR_DIMENSIONOVFLOW	5	Druck-Grafikgröße zu hoch
PDERR_INTERNALMEMORY	6	Kein Speicher für interne Variablen
PDERR_BUFFERMEMORY	7	Kein Speicher für Druckpuffer

IOPrtCmdReq (Printer Command Request)

```
00   ds.b  iopcr_IORequest,32   ; Eine IORequest-Struktur
32   dc.w  iopcr_PrtCommand     ; Steuerkommando
34   dc.b  iopcr_Parm0          ; Erster Parameter (falls nötig)
35   dc.b  iopcr_Parm1          ; Zweiter Parameter (falls nötig)
```

```
36      dc.b    iopcr_Parm2         ; Dritter Parameter (nicht benutzt)
37      dc.b    iopcr_Parm3         ; Vierter Parameter (nicht benutzt)
38              iopcr_SIZEOF
```

Kommando	Wert	ESC-Sequenz	Bedeutung
aRIS	0	ESCc	Drucker-Reset ausführen
aRIN	1	ESC#1	Drucker initialisieren
aIND	2	ESCD	Zeilenvorschub
aNEL	3	ESCE	Wagenrücklauf + Zeilenvorschub
aRI	4	ESCM	Zeile nach oben
aSGR0	5	ESC[0m	Standard-Zeichensatz
aSGR3	6	ESC[3m	Kursivschrift ein
aSGR23	7	ESC[23m	Kursivschrift aus
aSGR4	8	ESC[4m	Unterstrichen ein
aSGR24	9	ESC[24m	Unterstrichen aus
aSGR1	10	ESC[1m	Fettdruck ein
aSGR22	11	ESC[11m	Fettdruck aus
aSFC	12	ESC[3nm	Vordergrundfarbe n (0-9) einstellen
aSBC	13	ESC[4nm	Hintergrundfarbe n (0-9) einstellen
aSHORP0	14	ESC[0w	Normale Druckbreite
aSHORP2	15	ESC[2w	Elite-Druckdichte ein
aSHORP1	16	ESC[1w	Elite-Druckdichte aus
aSHORP4	17	ESC[4w	Schmalschrift ein
aSHORP3	18	ESC[3w	Schmalschrift aus
aSHORP6	19	ESC[6w	Breitschrift ein
aSHORT5	20	ESC[5w	Breitschrift aus
aDEN6	21	ESC[6"z	Schattendruck ein
aDEN5	22	ESC[5"z	Schattendruck aus
aDEN4	23	ESC[4"z	Doppeldruck ein
aDEN3	24	ESC[3"z	Doppeldruck aus
aDEN2	25	ESC[2"z	NLQ ein
aDEN1	26	ESC[1"z	NLQ aus
aSUS2	27	ESC[2v	Hochstellen ein
aSUS1	28	ESC[1v	Hochstellen aus
aSUS4	29	ESC[4v	Tiefstellen ein
aSUS3	30	ESC[3v	Tiefstellen aus
aSUS0	31	ESC[0v	Normale Schriftstellung
aPLU	32	ESCL	Teillinie aufwärts
aPLD	33	ESCK	Teillinie abwärts
aFNT0	34	ESC(B	Zeichensatz USA
aFNT1	35	ESC(R	Zeichensatz Frankreich
aFNT2	36	ESC(K	Zeichensatz Deutschland
aFNT3	37	ESC(A	Zeichensatz England
aFNT4	38	ESC(E	Zeichensatz Dänemark 1
aFNT5	39	ESC(H	Zeichensatz Schweden
aFNT6	40	ESC(Y	Zeichensatz Italien
aFNT7	41	ESC(Z	Zeichensatz Spanien
aFNT8	42	ESC(J	Zeichensatz Japan
aFNT9	43	ESC(6	Zeichensatz Norwegen
aFNT10	44	ESC(C	Zeichensatz Dänemark 2
aPROP2	45	ESC[2p	Proportional ein
aPROP1	46	ESC[1p	Proportional aus
aPROP0	47	ESC[0p	Proportional-Einstellung löschen
aTSS	48	ESC[n E	Proportional-Offset n einstellen
aJFY5	49	ESC[5 F	Linksausrichtung

aJFY7	50	ESC[7 F	Rechtsausrichtung
aJFY6	51	ESC[6 F	Zentrierte Ausrichtung
aJFY0	52	ESC[0 F	Ausrichtung aus
aJFY2	53	ESC[2 F	Wortabstand (Zentrierung)
aJFY3	54	ESC[3 F	Buchstabenabstand (Ausrichtung)
aVERP0	55	ESC[0z	Zeilenabstand 1/8"
aVERP1	56	ESC[1z	Zeilenabstand 1/6"
aSLPP	57	ESC[nt	Seitenlänge n einstellen
aPERF	58	ESC[nq	Überspringe n (n>0) Zeilen
aPERF0	59	ESC[0q	Überspringen aus
aLMS	60	ESC#9	Linker Rand gesetzt
aRMS	61	ESC#0	Rechter Rand gesetzt
aTMS	62	ESC#8	Oberer Rand gesetzt
aBMS	63	ESC#2	Unterer Rand gesetzt
aSTBM	64	ESC[Pn;mr	Setze Ränder auf n oben, m unten
aSLRM	65	ESC[Pn;ms	Setze Ränder auf n links, m rechts
aCAM	66	ESC#3	Ränder löschen
aHTS	67	ESCH	Horizontal-Tabulator setzen
aVTS	68	ESCJ	Vertikal-Tabulator setzen
aTBC0	69	ESC[0g	Horizontal-Tabulator löschen
aTBC3	70	ESC[3g	Alle Horizontal-Tabulatoren löschen
aTBC1	71	ESC[1g	Vertikal-Tabulator löschen
aTBC4	72	ESC[4g	Alle Vertikal-Tabulatoren löschen
aTBCALL	73	ESC#4	Alle V- und H-Tabulatoren löschen
aTBSALL	74	ESC#5	Standard-Tabulatoren setzen
aEXTEND	75	ESC[Pn"x	Erweitertes Kommando n ausführen

IODRPReq (Dump RastPort Request)

```
00   ds.b   iodrpr_IORequest     ; IORequest-Struktur
32   dc.l  *iodrpr_RastPort      ; Zeiger auf zu druckenden Rastport
36   dc.l  *iodrpr_ColorMap      ; Zeiger auf Farbtabelle des RPort
40   dc.l   iodrpr_Modes         ; View-Modus der Grafik
44   dc.w   iodrpr_SrcX          ; x-Start der Rastportgrafik
46   dc.w   iodrpr_SrcY          ; y-Start der Rastportgrafik
48   dc.w   iodrpr_SrcWidth      ; Breite der Rastportgrafik
50   dc.w   iodrpr_SrcHeight     ; Höhe der Rastportgrafik
52   dc.l   iodrpr_DestCols      ; Breite der Druckergrafik
56   dc.l   iodrpr_DestRows      ; Höhe der Druckergrafik
60   dc.w   iodrpr_Special       ; Optionsflags
62          iodrpr_SIZEOF
```

RPortDump-Flag	Wert	Bedeutung
SPECIAL_MILCOLS	$001	DestCols ist in 1/1000" angegeben
SPECIAL_MILROWS	$002	DestRows ist in 1/1000" angegeben
SPECIAL_FULLCOLS	$004	Maximale Spaltenzahl beim Druck
SPECIAL_FULLROWS	$008	Maximale Zeilenzahl beim Druck
SPECIAL_FRACCOLS	$010	DestCols ist ein Bruchteil von FULLCOLS
SPECIAL_FRACROWS	$020	DestRows ist ein Bruchteil von FULLROWS
SPECIAL_CENTER	$040	Zentrierter Ausdruck
SPECIAL_ASPECT	$080	Verhältnis Breite-Höhe korrigieren
SPECIAL_DENSITY1	$100	Minimale Druckdichte
SPECIAL_DENSITY2	$200	Zweitniedrigste Druckdichte

```
SPECIAL_DENSITY3      $300    Zweithöchste Druckdichte
SPECIAL_DENSITY4      $400    Höchste Druckdichte
```

G.6.3 Console-Device

ConUnit

```
000    dc.l    *mp_Succ             ;┐
004    dc.l    *mp_Pred             ;│
008    dc.b     mp_Type             ;│ MessagePort
009    dc.b     mp_Pri              ;│
010    dc.l    *mp_Name             ;│
014    dc.b     mp_Flags            ;│
015    dc.b     mp_SigBit           ;│
016    dc.l     mp_SigTask          ;│
020    dc.l    *lh_Head             ;│
024    dc.l    *lh_Tail             ;│
028    dc.l    *lh_TailPred         ;│
032    dc.b     lh_Type             ;│
033    dc.b     lh_Pad              ;┘

034    dc.l    *cu_Window           ; Zeiger auf Fenster
038    dc.w     cu_XCP              ;┬─ Zeichenposition
040    dc.w     cu_YCP              ;┘
042    dc.w     cu_XMax             ;┬─ Max. Zeichenposition
044    dc.w     cu_YMax             ;┘
046    dc.w     cu_XRSize           ;┬─ Größe des Zeichenrasters
048    dc.w     cu_YRSize           ;┘
050    dc.w     cu_XROrigin         ;┬─ Anfangspunkt
052    dc.w     cu_YROrigin         ;┘
054    dc.w     cu_XRExtant         ;┬─ Ausdehnung
056    dc.w     cu_YRExtant         ;┘
058    dc.w     cu_XMinShrink       ;┬─ Min. Fensterausdehnung
060    dc.w     cu_YMinShrink       ;┘
062    dc.w     cu_XCCP             ;┬─ Position des Cursors
064    dc.w     cu_YCCP             ;┘

066    dc.l    *km_LoKeyMapTypes    ;┐
070    dc.l    *km_LoKeyMap         ;│
074    dc.l    *km_LoCapsable       ;│
078    dc.l    *km_LoRepeatable     ;│   KeyMap-Struktur
082    dc.l    *km_HiKeyMapTypes    ;│     (aktuelle)
086    dc.l    *km_HiKeyMap         ;│
090    dc.l    *km_HiCapsable       ;│
094    dc.l    *km_HiRepeatable     ;┘

098    ds.w     cu_TabStops,80      ; Tabulatoren
258    dc.b     cu_Mask             ;┐
259    dc.b     cu_FgPen            ;│
260    dc.b     cu_BgPen            ;│
261    dc.b     cu_AOLPen           ;│
262    dc.b     cu_DrawMode         ;│
263    dc.b     cu_AreaPtSz         ;│
264    dc.l     cu_AreaPtrn         ;│
```

```
268     ds.b    cu_Minterms,8          ;┐ Rastport-Attribute
276     dc.l   *cu_Font                ;│
280     dc.b    cu_AlgoStyle           ;│
281     dc.b    cu_TxFlags             ;│
282     dc.w    cu_TxHeight            ;│
284     dc.w    cu_TxWidth             ;│
286     dc.w    cu_TxBaseline          ;│
288     dc.w    cu_TxSpacing           ;┘

290     ds.b    cu_Modes,3             ;┐ Modes und
293     ds.b    cu_RawEvents,3         ;┘ RawEvents
296             cu_SIZEOF
```

KeyMap

```
00      dc.l   *km_LoKeyMapTypes       ;┐ Zeiger auf Typentab.
04      dc.l   *km_LoKeyMap            ;│ Lo Zeiger auf KeyMap
08      dc.l   *km_LoCapsable          ;│ $00-$3f Capsable-Werte
12      dc.l   *km_LoRepeatable        ;┘ Repeatable-Werte

16      dc.l   *km_HiKeyMapTypes       ;┐ Zeiger auf Typentab.
20      dc.l   *km_HiKeyMap            ;│ Hi Zeiger auf KeyMap
24      dc.l   *km_HiCapsable          ;│ $40-$67 Capsable-Werte
28      dc.l   *km_HiRepeatable        ;┘ Repeatable-Werte
32              km_SIZEOF
```

Name	Wert	Bedeutung
KC_NORMAL	$00	Taste hat nur eine Belegung
KC_SHIFT	$01	Shift-Taste
KC_ALT	$02	Alt-Taste
KC_CONTROL	$04	Ctrl-Taste
KC_VANILLA	$07	Ctrl-Taste wird normal behandelt
KC_DOWNUP	$08	Taste wurde losgelassen
KC_DEAD	$20	keine Reaktion
KC_STRING	$40	Taste gibt eine Zeichenkette aus

G.6.4 Narrator-Device

Narrator-RB

```
00      ds.b    IOStdReq,48            ; Standard IORequest-Struktur
48      dc.w    rate                   ; Worte pro Minute
50      dc.w    pitch                  ; Frequez in Hz
52      dc.w    mode                   ; Betonungsmodus
54      dc.w    sex                    ; Geschlecht der Stimme
56      dc.l   *ch_masks               ; Zeiger auf Channel-Masks
60      dc.w    nm_masks               ; Anzahl der Masken
62      dc.w    volume                 ; Lautstärke
64      dc.w    sampfreq               ; Abtastrate
66      dc.b    mouths                 ; Mund-Flag
67      dc.b    chanmask               ;┐ Kanalmasken
68      dc.b    numchan                ;┘ (intern benutzt)
```

```
69      dc.b    pad                     ; Füllbyte
70              SIZEOF
```

Name	Wert	Bedeutung
ND_NotUsed	-01	nicht benutzt
ND_NoMem	-02	Speicher konnte nicht belegt werden
ND_NoAudLib	-03	Audio-Device konnte nicht geöffnet werden
ND_MakeBad	-04	Fehler beim Erstellen der Library
ND_UnitErr	-05	angegebene Unit != 0
ND_CantAlloc	-06	Audiokanäle konnten nicht belegt werden
ND_Unimpl	-07	falsches Kommando
ND_NoWrite	-08	"mouth shape" gelesen ohne zu schreiben
ND_Expunged	-09	Fehler beim Öffnen
ND_PhonErr	-20	Fehler bei Aussprache der Lautschrift
ND_RateErr	-21	Fehler durch rate-Wert
ND_Pitch	-22	Fehler durch pitch-Wert
ND_Sex	-23	Fehler durch sex-Einstellung
ND_ModeErr	-24	Fehler durch mode-Einstellung
ND_FreqErr	-25	Fehler durch sampfreq-Wert
ND_VolErr	-26	Fehler durch volume-Wert

Mouth_RB

```
00      ds.b    narrator_rb             ; normale Narr-IOReq-Struktur
70      dc.b    width                   ; Breite
71      dc.b    height                  ; Höhe
72      dc.b    shape                   ; Höhe/Breite
73      dc.b    pad                     ; Füllbyte
74              SIZEOF
```

Anhang H
Tabelle aller Library-Routinen

Nach Alphabet sortiert

Nach Offset sortiert

H.1 Die CList-Library (alphabetisch)

```
 -36  -$024  AllocCList (a1:cLPool)
-156  -$09c  ConcatCList (a0:sourceCList,a1:destCList)
-144  -$090  CopyCList (a0:cList)
 -48  -$030  FlushCList (a0:cList)
 -42  -$02a  FreeCList (a0:cList)
-114  -$072  GetCLBuf (a0:cList,a1:buffer,d1:length)
 -66  -$042  GetCLChar (a0:cList)
 -90  -$05a  GetCLWord (a0:cList)
-126  -$07e  IncrCLMark (a0:cList)
 -30  -$01e  InitCLPool (a0:cLPool,d0:size)
-120  -$078  MarkCList (a0:cList,d0:offset)
-132  -$084  PeekCLMark (a0:cList)
-108  -$06c  PutCLBuf (a0:cList,a1:buffer,d1:length)
 -60  -$03c  PutCLChar (a0:cList,d0:byte)
 -84  -$054  PutCLWord (a0:cList,d0:word)
 -54  -$036  SizeCList (a0:cList)
-138  -$08a  SplitCList (a0:cList)
-150  -$096  SubCList (a0:cList,d0:index,d1:length)
 -72  -$048  UnGetCLChar (a0:cList,d0:byte)
 -96  -$060  UnGetCLWord (a0:cList,d0:word)
 -78  -$04e  UnPutCLChar (a0:cList)
-102  -$066  UnPutCLWord (a0:cList)
```

H.2 Die CList-Library (nach Offsets)

```
 -30  -$01e  InitCLPool (a0:cLPool,d0:size)
 -36  -$024  AllocCList (a1:cLPool)
 -42  -$02a  FreeCList (a0:cList)
 -48  -$030  FlushCList (a0:cList)
 -54  -$036  SizeCList (a0:cList)
 -60  -$03c  PutCLChar (a0:cList,d0:byte)
 -66  -$042  GetCLChar (a0:cList)
 -72  -$048  UnGetCLChar (a0:cList,d0:byte)
 -78  -$04e  UnPutCLChar (a0:cList)
 -84  -$054  PutCLWord (a0:cList,d0:word)
 -90  -$05a  GetCLWord (a0:cList)
 -96  -$060  UnGetCLWord (a0:cList,d0:word)
-102  -$066  UnPutCLWord (a0:cList)
-108  -$06c  PutCLBuf (a0:cList,a1:buffer,d1:length)
-114  -$072  GetCLBuf (a0:cList,a1:buffer,d1:length)
-120  -$078  MarkCList (a0:cList,d0:offset)
-126  -$07e  IncrCLMark (a0:cList)
-132  -$084  PeekCLMark (a0:cList)
-138  -$08a  SplitCList (a0:cList)
-144  -$090  CopyCList (a0:cList)
-150  -$096  SubCList (a0:cList,d0:index,d1:length)
-156  -$09c  ConcatCList (a0:sourceCList,a1:destCList)
```

H.3 Die Console-Library (alphabetisch)

```
 -42  -$02a  CDInputHandler (a0:events,a1:device)
 -48  -$030  RawKeyConvert (a0:events,a1:buffer,d1:length,a2:keyMap)
```

H.4 Die Console-Library (nach Offsets)

```
 -42  -$02a  CDInputHandler (a0:events,a1:device)
 -48  -$030  RawKeyConvert (a0:events,a1:buffer,d1:length,a2:keyMap)
```

H.5 Die Diskfont-Library (alphabetisch)

```
 -36  -$024  AvailFonts (a0:buffer,d0:bufBytes,d1:Flags)
 -48  -$030  DisposeFontContents (a1:fontContentsHeader)
 -42  -$02a  NewFontContents (a0:fontsLock,a1:fontName)
 -30  -$01e  OpenDiskFont (a0:textAttr)
```

H.6 Die Diskfont-Library (nach Offset)

```
 -30  -$01e  OpenDiskFont (a0:textAttr)
 -36  -$024  AvailFonts (a0:buffer,d0:bufBytes,d1:Flags)
 -42  -$02a  NewFontContents (a0:fontsLock,a1:fontName)
 -48  -$030  DisposeFontContents (a1:fontContentsHeader)
```

H.7 Die DOS-Library (alphabetisch)

```
 -36  -$024  Close (d1:file)
-120  -$078  CreateDir (d1:name)
-138  -$08a  CreateProc (d1:name,d2:pri,d3:segList,d4:stack)
-126  -$07e  CurrentDir (d1:lock)
-192  -$0c0  DateStamp (d1:date)
-198  -$0c6  Delay (d1:timeout)
 -72  -$048  DeleteFile (d1:name)
-174  -$0ae  DeviceProc (d1:name)
 -96  -$060  DupLock (d1:lock)
```

```
-102  -$066  Examine (d1:lock,d2:fileInfoBlock)
-222  -$0de  Execute (d1:string,d2:file,d3:file)
-144  -$090  Exit (d1:returnCode)
-108  -$06c  ExNext (d1:lock,d2:fileInfoBlock)
-162  -$0a2  GetPacket (d1:wait)
-114  -$072  Info (d1:lock,d2:parameter)
 -54  -$036  Input ()
-132  -$084  IoErr ()
-216  -$0d8  IsInteractive (d1:file)
-150  -$096  LoadSeg (d1:filename)
 -84  -$054  Lock (d1:name,d2:type)
 -30  -$01e  Open (d1:name,d2:mode)
 -60  -$03c  Output ()
-210  -$0d2  ParentDir (d1:lock)
-168  -$0a8  QueuePacket (d1:packet)
 -42  -$02a  Read (d1:file,d2:buffer,d3:length)
 -78  -$04e  Rename (d1:oldname,d2:newname)
 -66  -$042  Seek (d1:file,d2:pos,d3:offset)
-180  -$0b4  SetComment (d1:name,d2:comment)
-186  -$0ba  SetProtection (d1:name,d2:mask)
-156  -$09c  UnLoadSeg (d1:segment)
 -90  -$05a  UnLock (d1:lock)
-204  -$0cc  WaitForChar (d1:file,d2:timeout)
 -48  -$030  Write (d1:file,d2:buffer,d3:length)
```

H.8 Die DOS-Library (nach Offsets)

```
 -30  -$01e  Open (d1:name,d2:mode)
 -36  -$024  Close (d1:file)
 -42  -$02a  Read (d1:file,d2:buffer,d3:length)
 -48  -$030  Write (d1:file,d2:buffer,d3:length)
 -54  -$036  Input ()
 -60  -$03c  Output ()
 -66  -$042  Seek (d1:file,d2:pos,d3:offset)
 -72  -$048  DeleteFile (d1:name)
 -78  -$04e  Rename (d1:oldname,d2:newname)
 -84  -$054  Lock (d1:name,d2:type)
 -90  -$05a  UnLock (d1:lock)
 -96  -$060  DupLock (d1:lock)
-102  -$066  Examine (d1:lock,d2:fileInfoBlock)
-108  -$06c  ExNext (d1:lock,d2:fileInfoBlock)
-114  -$072  Info (d1:lock,d2:parameter)
-120  -$078  CreateDir (d1:name)
-126  -$07e  CurrentDir (d1:lock)
-132  -$084  IoErr ()
-138  -$08a  CreateProc (d1:name,d2:pri,d3:segList,d4:stack)
-144  -$090  Exit (d1:returnCode)
-150  -$096  LoadSeg (d1:filename)
-156  -$09c  UnLoadSeg (d1:segment)
-162  -$0a2  GetPacket (d1:wait)
-168  -$0a8  QueuePacket (d1:packet)
-174  -$0ae  DeviceProc (d1:name)
```

```
-180  -$0b4  SetComment (d1:name,d2:comment)
-186  -$0ba  SetProtection (d1:name,d2:mask)
-192  -$0c0  DateStamp (d1:date)
-198  -$0c6  Delay (d1:timeout)
-204  -$0cc  WaitForChar (d1:file,d2:timeout)
-210  -$0d2  ParentDir (d1:lock)
-216  -$0d8  IsInteractive (d1:file)
-222  -$0de  Execute (d1:string,d2:file,d3:file)
```

H.9 Die Exec-Library (alphabetisch)

```
-480  -$1e0  AbortIO (a1:ioRequest)
-432  -$1b0  AddDevice (a1:device)
-240  -$0f0  AddHead (a0:list,a1:node)
-168  -$0a8  AddIntServer (d0:intNumber,a1:interrupt)
-396  -$18c  AddLibrary (a1:library)
-618  -$26a  AddMemList (d0:size,d1:attributes,d2:pri,a0:base,a1:name)
-354  -$162  AddPort (a1:port)
-486  -$1e6  AddResource (a1:resource)
-600  -$258  AddSemaphore (a1:sigSem)
-246  -$0f6  AddTail (a0:list,a1:node)
-282  -$11a  AddTask (a1:task,a2:initialPC,a3:finalPC)
-108  -$06c  Alert (d7:alertNum,a5:parameters)
-204  -$0cc  AllocAbs (d0:byteSize,a1:location)
-186  -$0ba  Allocate (a0:freelist,d0:byteSize)
-222  -$0de  AllocEntry (a0:entry)
-198  -$0c6  AllocMem (d0:byteSize,d1:requirements)
-330  -$14a  AllocSignal (d0:signalNum)
-342  -$156  AllocTrap (d0:trapNum)
-576  -$240  AttemptSemaphore (a0:sigSem)
-216  -$0d8  AvailMem (d1:requirements)
-180  -$0b4  Cause (a1:interrupt)
-468  -$1d4  CheckIO (a1:ioRequest)
-450  -$1c2  CloseDevice (a1:ioRequest)
-414  -$19e  CloseLibrary (a1:library)
-624  -$270  CopyMem (a0:source,a1:dest,d0:size)
-630  -$276  CopyMemQuick (a0:source,a1:dest,d0:size)
-192  -$0c0  Deallocate (a0:freelist,a1:memoryBlock,d0:byteSize)
-114  -$072  Debug ()
-120  -$078  Disable ()
 -60  -$03c  Dispatch ()
-456  -$1c8  DoIO (a1:ioRequest)
-126  -$07e  Enable ()
-270  -$10e  Enqueue (a0:list,a1:node)
 -66  -$042  Exception ()
 -36  -$024  ExitIntr ()
-276  -$114  FindName (a0:list,a1:name)
-390  -$186  FindPort (a1:name)
 -96  -$060  FindResident (a1:name)
-594  -$252  FindSemaphore (a1:sigSem)
-294  -$126  FindTask (a1:name)
-132  -$084  Forbid ()
```

```
-228  -$0e4  FreeEntry (a0:entry)
-210  -$0d2  FreeMem (a1:memoryBlock,d0:byteSize)
-336  -$150  FreeSignal (d0:signalNum)
-348  -$15c  FreeTrap (d0:trapNum)
-528  -$210  GetCC ()
-372  -$174  GetMsg (a0:port)
 -72  -$048  InitCode (d0:startClass,d1:version)
-102  -$066  InitResident (a1:resident,d1:segList)
-558  -$22e  InitSemaphore (a0:sigSem)
 -78  -$04e  InitStruct (a1:initTable,a2:memory,d0:size)
-234  -$0ea  Insert (a0:list,a1:node,a2:pred)
 -90  -$05a  MakeFunctions (a0:target,a1:functionArray,a2:funcDispBase)
 -84  -$054  MakeLibrary (a0:funcInit,a1:structInit,a2:libInit
                          d0:data,d1:code)
-564  -$234  ObtainSemaphore (a0:sigSem)
-582  -$246  ObtainSemaphoreList (a0:sigSem)
-408  -$198  OldOpenLibrary (a1:libName)
-444  -$1bc  OpenDevice (a0:devName,d0:unit,a1:ioRequest,d1:flags)
-552  -$228  OpenLibrary (a1:libName,d0:version)
-498  -$1f2  OpenResource (a1:resName,d0:version)
-138  -$08a  Permit ()
-540  -$21c  Procure (a0:semaport,a1:bidMsg)
-366  -$16e  PutMsg (a0:port,a1:message)
-522  -$20a  RawDoFmt (a0:format,a1:data,a2:putProc,a3:putData)
-504  -$1f8  RawIOInit ()
-510  -$1fe  RawMayGetChar ()
-516  -$204  RawPutChar (d0:char)
-570  -$23a  ReleaseSemaphore (a0:sigSem)
-588  -$24c  ReleaseSemaphoreList (a0:sigSem)
-438  -$1b6  RemoveDevice (a1:device)
-258  -$102  RemHead (a0:list)
-174  -$0ae  RemIntServer (d0:intNumber,a1:interrupt)
-402  -$192  RemLibrary (a1:library)
-252  -$0fc  Remove (a1:node)
-360  -$168  RemPort (a1:port)
-492  -$1ec  RemResource (a1:resource)
-606  -$25e  RemSemaphore (a1:sigSem)
-264  -$108  RemTail (a0:list)
-288  -$120  RemTask (a1:task)
-378  -$17a  ReplyMsg (a1:message)
 -48  -$030  Reschedule ()
 -42  -$02a  Schedule ()
-462  -$1ce  SendIO (a1:ioRequest)
-312  -$138  SetExcept (d0:newSignals,d1:signalSet)
-420  -$1a4  SetFunction (a1:library,a0:funcOffset,d0:funcEntry)
-162  -$0a2  SetIntVector (d0:intNumber,a1:interrupt)
-306  -$132  SetSignal (d0:newSignals,d1:signalSet)
-144  -$090  SetSR (d0:newSR,d1:mask)
-300  -$12c  SetTaskPri (a1:task,d0:priority)
-324  -$144  Signal (a1:task,d0:signalSet)
-612  -$264  SumKickData ()
-426  -$1aa  SumLibrary (a1:library)
-150  -$096  SuperState ()
 -30  -$01e  Supervisor ()
 -54  -$036  Switch ()
-534  -$216  TypeOfMem (a1:address)
```

```
-156  -$09c  UserState (a0:oldSysStack)
-546  -$222  Vacate (a0:semaport)
-318  -$13e  Wait (d0:signalSet)
-474  -$1da  WaitIO (a1:ioRequest)
-384  -$180  WaitPort (a0:port)
```

H.10 Die Exec-Library (nach Offsets)

```
 -30  -$01e  Supervisor ()
 -36  -$024  ExitIntr ()
 -42  -$02a  Schedule ()
 -48  -$030  Reschedule ()
 -54  -$036  Switch ()
 -60  -$03c  Dispatch ()
 -66  -$042  Exception ()
 -72  -$048  InitCode (d0:startClass,d1:version)
 -78  -$04e  InitStruct (a1:initTable,a2:memory,d0:size)
 -84  -$054  MakeLibrary (a0:funcInit,a1:structInit,a2:libInit
                          d0:data,d1:code)
 -90  -$05a  MakeFunctions (a0:target,a1:functionArray,a2:funcDispBase)
 -96  -$060  FindResident (a1:name)
-102  -$066  InitResident (a1:resident,d1:segList)
-108  -$06c  Alert (d7:alertNum,a5:parameters)
-114  -$072  Debug ()
-120  -$078  Disable ()
-126  -$07e  Enable ()
-132  -$084  Forbid ()
-138  -$08a  Permit ()
-144  -$090  SetSR (d0:newSR,d1:mask)
-150  -$096  SuperState ()
-156  -$09c  UserState (a0:oldSysStack)
-162  -$0a2  SetIntVector (d0:intNumber,a1:interrupt)
-168  -$0a8  AddIntServer (d0:intNumber,a1:interrupt)
-174  -$0ae  RemIntServer (d0:intNumber,a1:interrupt)
-180  -$0b4  Cause (a1:interrupt)
-186  -$0ba  Allocate (a0:freelist,d0:byteSize)
-192  -$0c0  Deallocate (a0:freelist,a1:memoryBlock,d0:byteSize)
-198  -$0c6  AllocMem (d0:byteSize,d1:requirements)
-204  -$0cc  AllocAbs (d0:byteSize,a1:location)
-210  -$0d2  FreeMem (a1:memoryBlock,d0:byteSize)
-216  -$0d8  AvailMem (d1:requirements)
-222  -$0de  AllocEntry (a0:entry)
-228  -$0e4  FreeEntry (a0:entry)
-234  -$0ea  Insert (a0:list,a1:node,a2:pred)
-240  -$0f0  AddHead (a0:list,a1:node)
-246  -$0f6  AddTail (a0:list,a1:node)
-252  -$0fc  Remove (a1:node)
-258  -$102  RemHead (a0:list)
-264  -$108  RemTail (a0:list)
-270  -$10e  Enqueue (a0:list,a1:node)
-276  -$114  FindName (a0:list,a1:name)
-282  -$11a  AddTask (a1:task,a2:initialPC,a3:finalPC)
```

```
-288  -$120  RemTask (a1:task)
-294  -$126  FindTask (a1:name)
-300  -$12c  SetTaskPri (a1:task,d0:priority)
-306  -$132  SetSignal (d0:newSignals,d1:signalSet)
-312  -$138  SetExcept (d0:newSignals,d1:signalSet)
-318  -$13e  Wait (d0:signalSet)
-324  -$144  Signal (a1:task,d0:signalSet)
-330  -$14a  AllocSignal (d0:signalNum)
-336  -$150  FreeSignal (d0:signalNum)
-342  -$156  AllocTrap (d0:trapNum)
-348  -$15c  FreeTrap (d0:trapNum)
-354  -$162  AddPort (a1:port)
-360  -$168  RemPort (a1:port)
-366  -$16e  PutMsg (a0:port,a1:message)
-372  -$174  GetMsg (a0:port)
-378  -$17a  ReplyMsg (a1:message)
-384  -$180  WaitPort (a0:port)
-390  -$186  FindPort (a1:name)
-396  -$18c  AddLibrary (a1:library)
-402  -$192  RemLibrary (a1:library)
-408  -$198  OldOpenLibrary (a1:libName)
-414  -$19e  CloseLibrary (a1:library)
-420  -$1a4  SetFunction (a1:library,a0:funcOffset,d0:funcEntry)
-426  -$1aa  SumLibrary (a1:library)
-432  -$1b0  AddDevice (a1:device)
-438  -$1b6  RemoveDevice (a1:device)
-444  -$1bc  OpenDevice (a0:devName,d0:unit,a1:ioRequest,d1:flags)
-450  -$1c2  CloseDevice (a1:ioRequest)
-456  -$1c8  DoIO (a1:ioRequest)
-462  -$1ce  SendIO (a1:ioRequest)
-468  -$1d4  CheckIO (a1:ioRequest)
-474  -$1da  WaitIO (a1:ioRequest)
-480  -$1e0  AbortIO (a1:ioRequest)
-486  -$1e6  AddResource (a1:resource)
-492  -$1ec  RemResource (a1:resource)
-498  -$1f2  OpenResource (a1:resName,d0:version)
-504  -$1f8  RawIOInit ()
-510  -$1fe  RawMayGetChar ()
-516  -$204  RawPutChar (d0:char)
-522  -$20a  RawDoFmt (a0:format,a1:data,a2:putProc,a3:putData)
-528  -$210  GetCC ()
-534  -$216  TypeOfMem (a1:address)
-540  -$21c  Procure (a0:semaport,a1:bidMsg)
-546  -$222  Vacate (a0:semaport)
-552  -$228  OpenLibrary (a1:libName,d0:version)
-558  -$22e  InitSemaphore (a0:sigSem)
-564  -$234  ObtainSemaphore (a0:sigSem)
-570  -$23a  ReleaseSemaphore (a0:sigSem)
-576  -$240  AttemptSemaphore (a0:sigSem)
-582  -$246  ObtainSemaphoreList (a0:sigSem)
-588  -$24c  ReleaseSemaphoreList (a0:sigSem)
-594  -$252  FindSemaphore (a1:sigSem)
-600  -$258  AddSemaphore (a1:sigSem)
-606  -$25e  RemSemaphore (a1:sigSem)
-612  -$264  SumKickData ()
-618  -$26a  AddMemList (d0:size,d1:attributes,d2:pri,a0:base,a1:name)
```

```
-624  -$270  CopyMem (a0:source,a1:dest,d0:size)
-630  -$276  CopyMemQuick (a0:source,a1:dest,d0:size)
```

H.11 Die Graphics-Library (alphabetisch)

```
-156  -$09c  AddAnimObj (a0:obj,a1:animationKey,a2:rastPort)
 -96  -$060  AddBob (a0:bob,a1:rastPort)
-480  -$1e0  AddFont (a1:textFont)
-102  -$066  AddVSprite (a0:vSprite,a1:rastPort)
-492  -$1ec  AllocRaster (d0:width,d1:height)
-504  -$1f8  AndRectRegion (a0:rgn,a1:rect)
-624  -$270  AndRegionRegion (a0:src,a1:dest)
-162  -$0a2  Animate (a0:animationKey,a1:rastPort)
-258  -$102  AreaDraw (a1:rastPort,d0:x,d1:y)
-186  -$0ba  AreaEllipse (a1:rastPort,d0:centerX,d1:centerY,d2:a,d3:b)
-264  -$108  AreaEnd (a1:rastPort)
-252  -$0fc  AreaMove (a1:rastPort,d0:x,d1:y)
-474  -$1da  AskFont (a1:rastPort,a0:textAttr)
 -84  -$054  AskSoftStyle (a1:rastPort)
-654  -$28e  AttemptLockLayerRom (a5:layer)
 -30  -$01e  BltBitMap (a0:srcBM,d0:srcX,d1:srcY,a1:destBM,d2:destX
                        d3:destY,d4:sizeX,d5:sizeY,d6:minTerm,d7:mask
                        a2:tempA)
-606  -$25e  BltBitMapRastPort (a0:srcBM,d0:srcX,d1:srcY,a1:destRP
                                d2:destX d3:destY,d4:SizeX,d5:SizeY
                                d6:minTerm)
-300  -$12c  BltClear (a1:memory,d0:size,d1:flags)
-636  -$27c  BltMaskBitMapRastPort (a0:srcBM,d0:srcX,d1:srcY,a1:destRP
                                    d2:destX d3:destY,d4:SizeX,d5:SizeY
                                    d6:minTerm,a2:bltMask)
-312  -$138  BltPattern (a1:rastPort,a0:ras,d0:x1,d1:y1,d2:maxX,d3:maxY
                         d4:fillBytes)
 -36  -$024  BltTemplate (a0:source,d0:srcX,d1:srcMod,a1:destRP,d2:destX
                          d3:destY,d4:sizeX,d5:sizeY)
-366  -$16e  CBump (a1:copperList)
-420  -$1a4  ChangeSprite (a0:viewPort,a1:simpleSprite,a2:data)
 -42  -$02a  ClearEOL (a1:rastPort)
-528  -$210  ClearRegion (a0:rgn)
 -48  -$030  ClearScreen (a1:rastPort)
-552  -$228  ClipBlit (a0:srcRP,d0:srcX,d1:srcY,a1:destRP,d2:destX
                       d3:destY,d4:sizeX,d5:sizeY,d6:minTerm)
 -78  -$04e  CloseFont (a1:textFont)
-372  -$174  CMove (a1:copperList,d0:destination,d1:data)
-450  -$1c2  CopySBitMap (a0:layer1,a1:layer2)
-378  -$17a  CWait (a1:copperList,d0:v,d1:h)
-462  -$1ce  DisownBlitter ()
-534  -$216  DisposeRegion (a0:rgn)
-108  -$06c  DoCollision (a1:rastPort)
-246  -$0f6  Draw (a1:rastPort,d0:x,d1:y)
-180  -$0b4  DrawEllipse (a1:rastPort,d0:centerX,d1:centerY,d2:a,d3:b)
-114  -$072  DrawGList (a1:rastPort,a0:viewPort)
-330  -$14a  Flood (a1:rastPort,d2:mode,d0:x,d1:y)
```

-576	-$240	FreeColorMap (a0:colormap)
-546	-$222	FreeCopList (a0:copList)
-564	-$234	FreeCprList (a0:cprList)
-600	-$258	FreeGBuffers (a0:animObj,a1:rastPort,d0:doubleBuffer)
-498	-$1f2	FreeRaster (a0:planePtr,d0:widht,d1:height)
-414	-$19e	FreeSprite (d0:num)
-540	-$21c	FreeVPortCopLists (a0:viewPort)
-570	-$23a	GetColorMap (d0:entries)
-168	-$0a8	GetGBuffers (a0:animationObj,a1:rastPort,d0:doubleBuffer)
-582	-$246	GetRGB4 (a0:colormap,d0:entry)
-408	-$198	GetSprite (a0:simpleSprite,d0:num)
-282	-$11a	InitArea (a0:areaInfo,a1:vectorTable,d0:tableSize)
-390	-$186	InitBitMap (a0:bitMap,d0:depth,d1:width,d2:height)
-120	-$078	InitGels (a0:dummyHead,a1:dummyTail,a2:gelsInfo)
-174	-$0ae	InitGMasks (a0:animationObj)
-126	-$07e	InitMasks (a0:vSprite)
-198	-$0c6	InitRastPort (a1:rastPort)
-468	-$1d4	InitTmpRas (a0:tmpRas,a1:buff,d0:size)
-360	-$168	InitView (a1:view)
-204	-$0cc	InitViewPort (a0:viewPort)
-192	-$0c0	LoadRGB4 (a0:viewPort,a1:colors,d0:count)
-222	-$0de	LoadView (a1:view)
-432	-$1b0	LockLayerRom (a5:layer)
-216	-$0d8	MakeVPort (a0:view,a1:viewPort)
-240	-$0f0	Move (a1:rastPort,d0:x,d1:y)
-426	-$1aa	MoveSprite (a0:viewPort,a2:simpleSprite,d0:x,d1:y)
-210	-$0d2	MrgCop (a1:view)
-516	-$204	NewRegion ()
-72	-$048	OpenFont (a0:textAttr)
-510	-$1fe	OrRectRegion (a0:rgn,a1:rect)
-612	-$264	OrRegionRegion (a0:src,a1:dest)
-456	-$1c8	OwnBlitter ()
-336	-$150	PolyDraw (a1:rastPort,d0:count,a0:polyTable)
-276	-$114	QBlit (a1:blit)
-294	-$126	QBSBlit ()
-318	-$13e	ReadPixel (a1:rastPort,d0:x,d1:y)
-306	-$132	RectFill (a1:rastPort,d0:xl,d1:yl,d2:xu,d3:yu)
-486	-$1e6	RemFont (a1:textFont)
-132	-$084	RemIBob (a0:bob,a1:rastPort,a2:viewPort)
-138	-$08a	RemVSprite (a0:vsprite)
-396	-$18c	ScrollRaster (a1:rastPort,d0:dX,d1:dY,d2:minX,d3:minY d4:maxX,d5:maxY)
-588	-$24c	ScrollVPort (a0:viewPort)
-342	-$156	SetAPen (a1:rastPort,d0:pen)
-348	-$15c	SetBPen (a1:rastPort,d0:pen)
-144	-$090	SetCollision (d0:type,a0:routine,a1:gelsInfo)
-354	-$162	SetDrMd (a1:rastPort,d0:drawMode)
-66	-$042	SetFont (a1:rastPort,a0:textFont)
-234	-$0ea	SetRast (a1:rastPort,d0:color)
-288	-$120	SetRGB4 (a0:viewPort,d0:index,d1:r,d2:g,d3:b)
-630	-$276	SetRGB4CM (a0:colorMap,d0:color,d1:r,d2:g,d3:b)
-90	-$05a	SetSoftStyle (a1:rastPort,d0:style,d1:enable)
-150	-$096	SortGList (a1:rastPort)
-444	-$1bc	SyncSBitMap (a0:layer)
-60	-$03c	Text (a1:rastPort,a0:string,d0:count)
-54	-$036	TextLength (a1:rastPort,a0:string,d0:count)

```
-594  -$252  UCopperListInit (a0:copperlist,d0:num)
-438  -$1b6  UnlockLayerRom (a5:layer)
-384  -$180  VBeamPos ()
-228  -$0e4  WaitBlit ()
-402  -$192  WaitBOVP (a0:viewPort)
-270  -$10e  WaitTOF ()
-324  -$144  WritePixel (a1:rastPort,d0:x,d1:y)
-558  -$22e  XorRectRegion (a0:rgn,a1:rect)
-618  -$26a  XorRegionRegion (a0:src,a1:dest)
```

H.12 Die Graphics-Library (nach Offsets)

```
 -30  -$01e  BltBitMap (a0:srcBM,d0:srcX,d1:srcY,a1:destBM,d2:destX
                        d3:destY,d4:sizeX,d5:sizeY,d6:minTerm,d7:mask
                        a2:tempA)
 -36  -$024  BltTemplate (a0:source,d0:srcX,d1:srcMod,a1:destRP,d2:destX
                          d3:destY,d4:sizeX,d5:sizeY)
 -42  -$02a  ClearEOL (a1:rastPort)
 -48  -$030  ClearScreen (a1:rastPort)
 -54  -$036  TextLength (a1:rastPort,a0:string,d0:count)
 -60  -$03c  Text (a1:rastPort,a0:string,d0:count)
 -66  -$042  SetFont (a1:rastPort,a0:textFont)
 -72  -$048  OpenFont (a0:textAttr)
 -78  -$04e  CloseFont (a1:textFont)
 -84  -$054  AskSoftStyle (a1:rastPort)
 -90  -$05a  SetSoftStyle (a1:rastPort,d0:style,d1:enable)
 -96  -$060  AddBob (a0:bob,a1:rastPort)
-102  -$066  AddVSprite (a0:vSprite,a1:rastPort)
-108  -$06c  DoCollision (a1:rastPort)
-114  -$072  DrawGList (a1:rastPort,a0:viewPort)
-120  -$078  InitGels (a0:dummyHead,a1:dummyTail,a2:gelsInfo)
-126  -$07e  InitMasks (a0:vSprite)
-132  -$084  RemIBob (a0:bob,a1:rastPort,a2:viewPort)
-138  -$08a  RemVSprite (a0:vsprite)
-144  -$090  SetCollision (d0:type,a0:routine,a1:gelsInfo)
-150  -$096  SortGList (a1:rastPort)
-156  -$09c  AddAnimObj (a0:obj,a1:animationKey,a2:rastPort)
-162  -$0a2  Animate (a0:animationKey,a1:rastPort)
-168  -$0a8  GetGBuffers (a0:animationObj,a1:rastPort,d0:doubleBuffer)
-174  -$0ae  InitGMasks (a0:animationObj)
-180  -$0b4  DrawEllipse (a1:rastPort,d0:centerX,d1:centerY,d2:a,d3:b)
-186  -$0ba  AreaEllipse (a1:rastPort,d0:centerX,d1:centerY,d2:a,d3:b)
-192  -$0c0  LoadRGB4 (a0:viewPort,a1:colors,d0:count)
-198  -$0c6  InitRastPort (a1:rastPort)
-204  -$0cc  InitViewPort (a0:viewPort)
-210  -$0d2  MrgCop (a1:view)
-216  -$0d8  MakeVPort (a0:view,a1:viewPort)
-222  -$0de  LoadView (a1:view)
-228  -$0e4  WaitBlit ()
-234  -$0ea  SetRast (a1:rastPort,d0:color)
-240  -$0f0  Move (a1:rastPort,d0:x,d1:y)
-246  -$0f6  Draw (a1:rastPort,d0:x,d1:y)
```

```
-252  -$0fc  AreaMove (a1:rastPort,d0:x,d1:y)
-258  -$102  AreaDraw (a1:rastPort,d0:x,d1:y)
-264  -$108  AreaEnd (a1:rastPort)
-270  -$10e  WaitTOF ()
-276  -$114  QBlit (a1:blit)
-282  -$11a  InitArea (a0:areaInfo,a1:vectorTable,d0:tableSize)
-288  -$120  SetRGB4 (a0:viewPort,d0:index,d1:r,d2:g,d3:b)
-294  -$126  QBSBlit ()
-300  -$12c  BltClear (a1:memory,d0:size,d1:flags)
-306  -$132  RectFill (a1:rastPort,d0:x1,d1:y1,d2:xu,d3:yu)
-312  -$138  BltPattern (a1:rastPort,a0:ras,d0:x1,d1:y1,d2:maxX,d3:maxY
                        d4:fillBytes)
-318  -$13e  ReadPixel (a1:rastPort,d0:x,d1:y)
-324  -$144  WritePixel (a1:rastPort,d0:x,d1:y)
-330  -$14a  Flood (a1:rastPort,d2:mode,d0:x,d1:y)
-336  -$150  PolyDraw (a1:rastPort,d0:count,a0:polyTable)
-342  -$156  SetAPen (a1:rastPort,d0:pen)
-348  -$15c  SetBPen (a1:rastPort,d0:pen)
-354  -$162  SetDrMd (a1:rastPort,d0:drawMode)
-360  -$168  InitView (a1:view)
-366  -$16e  CBump (a1:copperList)
-372  -$174  CMove (a1:copperList,d0:destination,d1:data)
-378  -$17a  CWait (a1:copperList,d0:v,d1:h)
-384  -$180  VBeamPos ()
-390  -$186  InitBitMap (a0:bitMap,d0:depth,d1:width,d2:height)
-396  -$18c  ScrollRaster (a1:rastPort,d0:dX,d1:dY,d2:minX,d3:minY
                          d4:maxX,d5:maxY)
-402  -$192  WaitBOVP (a0:viewPort)
-408  -$198  GetSprite (a0:simpleSprite,d0:num)
-414  -$19e  FreeSprite (d0:num)
-420  -$1a4  ChangeSprite (a0:viewPort,a1:simpleSprite,a2:data)
-426  -$1aa  MoveSprite (a0:viewPort,a2:simpleSprite,d0:x,d1:y)
-432  -$1b0  LockLayerRom (a5:layer)
-438  -$1b6  UnlockLayerRom (a5:layer)
-444  -$1bc  SyncSBitMap (a0:layer)
-450  -$1c2  CopySBitMap (a0:layer1,a1:layer2)
-456  -$1c8  OwnBlitter ()
-462  -$1ce  DisownBlitter ()
-468  -$1d4  InitTmpRas (a0:tmpRas,a1:buff,d0:size)
-474  -$1da  AskFont (a1:rastPort,a0:textAttr)
-480  -$1e0  AddFont (a1:textFont)
-486  -$1e6  RemFont (a1:textFont)
-492  -$1ec  AllocRaster (d0:width,d1:height)
-498  -$1f2  FreeRaster (a0:planePtr,d0:widht,d1:height)
-504  -$1f8  AndRectRegion (a0:rgn,a1:rect)
-510  -$1fe  OrRectRegion (a0:rgn,a1:rect)
-516  -$204  NewRegion ()
-528  -$210  ClearRegion (a0:rgn)
-534  -$216  DisposeRegion (a0:rgn)
-540  -$21c  FreeVPortCopLists (a0:viewPort)
-546  -$222  FreeCopList (a0:copList)
-552  -$228  ClipBlit (a0:srcRP,d0:srcX,d1:srcY,a1:destRP,d2:destX
                      d3:destY,d4:sizeX,d5:sizeY,d6:minTerm)
-558  -$22e  XorRectRegion (a0:rgn,a1:rect)
-564  -$234  FreeCprList (a0:cprList)
-570  -$23a  GetColorMap (d0:entries)
```

```
-576  -$240  FreeColorMap (a0:colormap)
-582  -$246  GetRGB4 (a0:colormap,d0:entry)
-588  -$24c  ScrollVPort (a0:viewPort)
-594  -$252  UCopperListInit (a0:copperlist,d0:num)
-600  -$258  FreeGBuffers (a0:animObj,a1:rastPort,d0:doubleBuffer)
-606  -$25e  BltBitMapRastPort (a0:srcBM,d0:srcX,d1:srcY,a1:destRP
                                d2:destX d3:destY,d4:SizeX,d5:SizeY
                                d6:minTerm)
-612  -$264  OrRegionRegion (a0:src,a1:dest)
-618  -$26a  XorRegionRegion (a0:src,a1:dest)
-624  -$270  AndRegionRegion (a0:src,a1:dest)
-630  -$276  SetRGB4CM (a0:colorMap,d0:color,d1:r,d2:g,d3:b)
-636  -$27c  BltMaskBitMapRastPort (a0:srcBM,d0:srcX,d1:srcY,a1:destRP
                                    d2:destX d3:destY,d4:SizeX,d5:SizeY
                                    d6:minTerm,a2:bltMask)
-654  -$28e  AttemptLockLayerRom (a5:layer)
```

H.13 Die Icon-Library (alphabetisch)

```
 -72  -$048  AddFreeList (a0:freelist,a1:mem,a2:size)
 -66  -$042  AllocWBObject ()
-108  -$06c  BumpRevision (a0:newname,a1:oldname)
 -96  -$060  FindToolType (a0:toolTypeArray,a1:typeName)
 -90  -$05a  FreeDiskObject (a0:diskobj)
 -54  -$036  FreeFreeList (a0:freelist)
 -60  -$03c  FreeWBObject (a0:WBObject)
 -78  -$04e  GetDiskObject (a0:name)
 -42  -$02a  GetIcon (a0:name,a1:icon,a2:freelist)
 -30  -$01e  GetWBObject (a0:name)
-102  -$066  MatchToolValue (a0:typeString,a1:value)
 -84  -$054  PutDiskObject (a0:name,a1:diskobj)
 -48  -$030  PutIcon (a0:name,a1:icon)
 -36  -$024  PutWBObject (a0:name,a1:object)
```

H.14 Die Icon-Library (nach Offsets)

```
 -30  -$01e  GetWBObject (a0:name)
 -36  -$024  PutWBObject (a0:name,a1:object)
 -42  -$02a  GetIcon (a0:name,a1:icon,a2:freelist)
 -48  -$030  PutIcon (a0:name,a1:icon)
 -54  -$036  FreeFreeList (a0:freelist)
 -60  -$03c  FreeWBObject (a0:WBObject)
 -66  -$042  AllocWBObject ()
 -72  -$048  AddFreeList (a0:freelist,a1:mem,a2:size)
 -78  -$04e  GetDiskObject (a0:name)
 -84  -$054  PutDiskObject (a0:name,a1:diskobj)
 -90  -$05a  FreeDiskObject (a0:diskobj)
 -96  -$060  FindToolType (a0:toolTypeArray,a1:typeName)
```

```
-102  -$066  MatchToolValue (a0:typeString,a1:value)
-108  -$06c  BumpRevision (a0:newname,a1:oldname)
```

H.15 Die Intuition-Library (alphabetisch)

```
-462  -$1ce  ActivateGadget (a0:Gadget,a1:Window,a2:Req)
-450  -$1c2  ActivateWindow (a0:Window)
 -42  -$02a  AddGadget (a0:AddPtr,a1:Gadget,d0:Position)
-438  -$1b6  AddGList (a0:AddPtr,a1:Gadget,d0:Pos,d1:NumGad,a2:Req)
-396  -$18c  AllocRemember (a0:RememberKey,d0:Size,d1:Flags)
-402  -$192  AlohaWorkbench (a0:wbport)
-348  -$15c  AutoRequest (a0:Win,a1:Body,a2:PText,a3:NText,d0:PFlag
                               d1:NFlag,d2:W,d3:H)
-354  -$162  BeginRefresh (a0:Window)
-360  -$168  BuildSysRequest (a0:Win,a1:Body,a2:PText,a3:NText,d0:Flag
                                   d1:W,d2:H)
 -48  -$030  ClearDMRequest (a0:Window)
 -54  -$036  ClearMenuStrip (a0:Window)
 -60  -$03c  ClearPointer (a0:Window)
 -66  -$042  CloseScreen (a0:Screen)
 -72  -$048  CloseWindow (a0:Window)
 -78  -$04e  CloseWorkBench ()
 -84  -$054  CurrentTime (a0:Seconds,a1:Micros)
 -90  -$05a  DisplayAlert (d0:AlertNumber,a0:String,a1:Height)
 -96  -$060  DisplayBeep (a0:Screen)
-102  -$066  DoubleClick (d0:sseconds,d1:smicros,d2:cseconds,d3:cmicros)
-108  -$06c  DrawBorder (a0:RPort,a1:Border,d0:LeftOffset,d1:TopOffset)
-114  -$072  DrawImage (a0:RPort,a1:Image,d0:LeftOffset,d1:TopOffset)
-366  -$16e  EndRefresh (a0:Window,d0:Complete)
-120  -$078  EndRequest (a0:requester,a1:Window)
-408  -$198  FreeRemember (a0:RememberKey,d0:ReallyForget)
-372  -$174  FreeSysRequest (a0:Window)
-126  -$07e  GetDefPrefs (a0:preferences,d0:size)
-132  -$084  GetPrefs (a0:preferences,d0:size)
-426  -$1aa  GetScreenData (a0:buffer,d0:size,d1:type:a1:Screen)
-138  -$08a  InitRequester (a0:Req)
-330  -$14a  IntuiTextLength (a0:IText)
 -36  -$024  Intuition (a0:ievent)
-144  -$090  ItemAddress (a0:MenuStrip,d0:MenuNumber)
-414  -$19e  LockIBase (d0:dontknow)
-378  -$17a  MakeScreen (a0:Screen)
-150  -$096  ModifyIDCMP (a0:Window,d0:Flags)
-156  -$09c  ModifyProp (a0:Gadget,a1:Win,a2:Req,d0:Flags,d1:HPos
                              d2:VPos,d3:HBody,d4:VBody)
-162  -$0a2  MoveScreen (a0:Screen,d0:dx,d1:dy)
-168  -$0a8  MoveWindow (a0:Window,d0:dx,d1:dy)
-468  -$1d4  NewModifyProp (a0:Gad,a1:Win,a2:Req,d0:Flag,d1:HPos
                                  d2:VPos,d3:HBod,d4:VBod,d5:NumGad)
-174  -$0ae  OffGadget (a0:Gadget,a1:Win,a2:Req)
-180  -$0b4  OffMenu (a0:Window,d0:MenuNum)
-186  -$0ba  OnGadget (a0:Gadget,a1:Win,a2:Req)
-192  -$0c0  OnMenu (a0:Window,d0:MenuNum)
```

-30	-$01e	OpenIntuition ()
-198	-$0c6	OpenScreen (a0:Args)
-204	-$0cc	OpenWindow (a0:Args)
-210	-$0d2	OpenWorkBench ()
-216	-$0d8	PrintIText (a0:rp,a1:IText,d0:LeftOffset,d1:TopOffset)
-222	-$0de	RefreshGadgets (a0:Gadgets,a1:Win,a2:Req)
-432	-$1b0	RefreshGList (a0:Gadgets,a1:Win,a2:Req,d0:NumGad)
-456	-$1c8	RefreshWindow (a0:Window)
-384	-$180	RemakeDisplay ()
-228	-$0e4	RemoveGadget (a0:RemPtr,a1:Gadget)
-444	-$1bc	RemoveGList (a0:RemPtr,a1:Gadget,d0:NumGad)
-234	-$0ea	ReportMouse (a0:Window,d0:Boolean)
-240	-$0f0	Request (a0:Req,a1:Win)
-390	-$186	RethinkDisplay ()
-246	-$0f6	ScreenToBack (a0:Screen)
-252	-$0fc	ScreenToFront (a0:Screen)
-258	-$102	SetDMRequest (a0:Win,a1:Req)
-264	-$108	SetMenuStrip (a0:Win,a1:Menu)
-270	-$10e	SetPointer (a0:Win,a1:Pointer,d0:Height,d1:Width,d2:HotX d3:HotY)
-324	-$144	SetPrefs (a0:preferences,d0:size,d1:flags)
-276	-$114	SetWindowTitles (a0:window,a1:winTitle,a2:scrTitle)
-282	-$11a	ShowTitle (a0:Screen,d0:ShowIt)
-288	-$120	SizeWindow (a0:Window,d0:dx,d1:dy)
-420	-$1a4	UnlockIBase (a0:IBLock)
-294	-$126	ViewAddress ()
-300	-$12c	ViewPortAddress (a0:Window)
-336	-$150	WBenchToBack ()
-342	-$156	WBenchToFront ()
-318	-$13e	WindowLimits (a0:Window,d0:WMin,d1:HMin,d2:WMax,d3:HMax)
-306	-$132	WindowToBack (a0:Window)
-312	-$138	WindowToFront (a0:Window)

H.16 Die Intuition-Library (nach Offsets)

-30	-$01e	OpenIntuition ()
-36	-$024	Intuition (a0:ievent)
-42	-$02a	AddGadget (a0:AddPtr,a1:Gadget,d0:Position)
-48	-$030	ClearDMRequest (a0:Window)
-54	-$036	ClearMenuStrip (a0:Window)
-60	-$03c	ClearPointer (a0:Window)
-66	-$042	CloseScreen (a0:Screen)
-72	-$048	CloseWindow (a0:Window)
-78	-$04e	CloseWorkBench ()
-84	-$054	CurrentTime (a0:Seconds,a1:Micros)
-90	-$05a	DisplayAlert (d0:AlertNumber,a0:String,a1:Height)
-96	-$060	DisplayBeep (a0:Screen)
-102	-$066	DoubleClick (d0:sseconds,d1:smicros,d2:cseconds,d3:cmicros)
-108	-$06c	DrawBorder (a0:RPort,a1:Border,d0:LeftOffset,d1:TopOffset)
-114	-$072	DrawImage (a0:RPort,a1:Image,d0:LeftOffset,d1:TopOffset)
-120	-$078	EndRequest (a0:requester,a1:Window)
-126	-$07e	GetDefPrefs (a0:preferences,d0:size)

```
-132  -$084  GetPrefs (a0:preferences,d0:size)
-138  -$08a  InitRequester (a0:Req)
-144  -$090  ItemAddress (a0:MenuStrip,d0:MenuNumber)
-150  -$096  ModifyIDCMP (a0:Window,d0:Flags)
-156  -$09c  ModifyProp (a0:Gadget,a1:Win,a2:Req,d0:Flags,d1:HPos
                         d2:VPos,d3:HBody,d4:VBody)
-162  -$0a2  MoveScreen (a0:Screen,d0:dx,d1:dy)
-168  -$0a8  MoveWindow (a0:Window,d0:dx,d1:dy)
-174  -$0ae  OffGadget (a0:Gadget,a1:Win,a2:Req)
-180  -$0b4  OffMenu (a0:Window,d0:MenuNum)
-186  -$0ba  OnGadget (a0:Gadget,a1:Win,a2:Req)
-192  -$0c0  OnMenu (a0:Window,d0:MenuNum)
-198  -$0c6  OpenScreen (a0:Args)
-204  -$0cc  OpenWindow (a0:Args)
-210  -$0d2  OpenWorkBench ()
-216  -$0d8  PrintIText (a0:rp,a1:IText,d0:LeftOffset,d1:TopOffset)
-222  -$0de  RefreshGadgets (a0:Gadgets,a1:Win,a2:Req)
-228  -$0e4  RemoveGadget (a0:RemPtr,a1:Gadget)
-234  -$0ea  ReportMouse (a0:Window,d0:Boolean)
-240  -$0f0  Request (a0:Req,a1:Win)
-246  -$0f6  ScreenToBack (a0:Screen)
-252  -$0fc  ScreenToFront (a0:Screen)
-258  -$102  SetDMRequest (a0:Win,a1:Req)
-264  -$108  SetMenuStrip (a0:Win,a1:Menu)
-270  -$10e  SetPointer (a0:Win,a1:Pointer,d0:Height,d1:Width,d2:HotX
                         d3:HotY)
-276  -$114  SetWindowTitles (a0:window,a1:winTitle,a2:scrTitle)
-282  -$11a  ShowTitle (a0:Screen,d0:ShowIt)
-288  -$120  SizeWindow (a0:Window,d0:dx,d1:dy)
-294  -$126  ViewAddress ()
-300  -$12c  ViewPortAddress (a0:Window)
-306  -$132  WindowToBack (a0:Window)
-312  -$138  WindowToFront (a0:Window)
-318  -$13e  WindowLimits (a0:Window,d0:WMin,d1:HMin,d2:WMax,d3:HMax)
-324  -$144  SetPrefs (a0:preferences,d0:size,d1:flags)
-330  -$14a  IntuiTextLength (a0:IText)
-336  -$150  WBenchToBack ()
-342  -$156  WBenchToFront ()
-348  -$15c  AutoRequest (a0:Win,a1:Body,a2:PText,a3:NText,d0:PFlag
                          d1:NFlag,d2:W,d3:H)
-354  -$162  BeginRefresh (a0:Window)
-360  -$168  BuildSysRequest (a0:Win,a1:Body,a2:PText,a3:NText,d0:Flag
                              d1:W,d2:H)
-366  -$16e  EndRefresh (a0:Window,d0:Complete)
-372  -$174  FreeSysRequest (a0:Window)
-378  -$17a  MakeScreen (a0:Screen)
-384  -$180  RemakeDisplay ()
-390  -$186  RethinkDisplay ()
-396  -$18c  AllocRemember (a0:RememberKey,d0:Size,d1:Flags)
-402  -$192  AlohaWorkbench (a0:wbport)
-408  -$198  FreeRemember (a0:RememberKey,d0:ReallyForget)
-414  -$19e  LockIBase (d0:dontknow)
-420  -$1a4  UnlockIBase (a0:IBLock)
-426  -$1aa  GetScreenData (a0:buffer,d0:size,d1:type:a1:Screen)
-432  -$1b0  RefreshGList (a0:Gadgets,a1:Win,a2:Req,d0:NumGad)
-438  -$1b6  AddGList (a0:AddPtr,a1:Gadget,d0:Pos,d1:NumGad,a2:Req)
```

```
-444  -$1bc  RemoveGList (a0:RemPtr,a1:Gadget,d0:NumGad)
-450  -$1c2  ActivateWindow (a0:Window)
-456  -$1c8  RefreshWindow (a0:Window)
-462  -$1ce  ActivateGadget (a0:Gadget,a1:Window,a2:Req)
-468  -$1d4  NewModifyProp (a0:Gad,a1:Win,a2:Req,d0:Flag,d1:HPos
                            d2:VPos,d3:HBod,d4:VBod,d5:NumGad)
```

H.17 Die Layers-Library (alphabetisch)

```
 -78  -$04e  BeginUpdate (a0:layer)
 -54  -$036  BehindLayer (a0:li,a1:layer)
 -42  -$02a  CreateBehindLayer (a0:li,a1:bm,d0:x0,d1:y0,d2:x1,d3:y1
                                d4:flags,a2:bm2)
 -36  -$024  CreateUpfrontLayer (a0:li,a1:bm,d0:x0,d1:y0,d2:x1,d3:y1
                                 d4:flags,a2:bm2)
 -90  -$05a  DeleteLayer (a0:li,a1:layer)
-150  -$096  DisposeLayerInfo (a0:li)
 -84  -$054  EndUpdate (a0:layer,d0:flag)
-156  -$09c  FattenLayerInfo (a0:li)
 -30  -$01e  InitLayers (a0:li)
-174  -$0ae  InstallClipRegion (a0:layer,a1:region)
 -96  -$060  LockLayer (a0:li,a1:layer)
-120  -$078  LockLayerInfo (a0:li)
-108  -$06c  LockLayers (a0:li)
 -60  -$03c  MoveLayer (a0:li,a1:layer,d0:dx,d1:dy)
-168  -$0a8  MoveLayerInFrontOf (a0:movelayer,a1:backlayer)
-144  -$090  NewLayerInfo ()
 -72  -$048  ScrollLayer (a0:li,a1:layer,d0:dx,d1:dy)
 -66  -$042  SizeLayer (a0:li,a1:layer,d0:dx,d1:dy)
-126  -$07e  SwapBitsRastPortClipRect (a0:rp,a1:cr)
-162  -$0a2  ThinLayerInfo (a0:li)
-102  -$066  UnlockLayer (a0:layer)
-138  -$08a  UnlockLayerInfo (a0:li)
-114  -$072  UnlockLayers (a0:li)
 -48  -$030  UpfrontLayer (a0:li,a1:layer)
-132  -$084  WhichLayer (a0:li,d0:x,d1:y)
```

H.18 Die Layers-Library (nach Offsets)

```
 -30  -$01e  InitLayers (a0:li)
 -36  -$024  CreateUpfrontLayer (a0:li,a1:bm,d0:x0,d1:y0,d2:x1,d3:y1
                                 d4:flags,a2:bm2)
 -42  -$02a  CreateBehindLayer (a0:li,a1:bm,d0:x0,d1:y0,d2:x1,d3:y1
                                d4:flags,a2:bm2)
 -48  -$030  UpfrontLayer (a0:li,a1:layer)
 -54  -$036  BehindLayer (a0:li,a1:layer)
 -60  -$03c  MoveLayer (a0:li,a1:layer,d0:dx,d1:dy)
 -66  -$042  SizeLayer (a0:li,a1:layer,d0:dx,d1:dy)
```

```
 -72  -$048  ScrollLayer (a0:li,a1:layer,d0:dx,d1:dy)
 -78  -$04e  BeginUpdate (a0:layer)
 -84  -$054  EndUpdate (a0:layer,d0:flag)
 -90  -$05a  DeleteLayer (a0:li,a1:layer)
 -96  -$060  LockLayer (a0:li,a1:layer)
-102  -$066  UnlockLayer (a0:layer)
-108  -$06c  LockLayers (a0:li)
-114  -$072  UnlockLayers (a0:li)
-120  -$078  LockLayerInfo (a0:li)
-126  -$07e  SwapBitsRastPortClipRect (a0:rp,a1:cr)
-132  -$084  WhichLayer (a0:li,d0:x,d1:y)
-138  -$08a  UnlockLayerInfo (a0:li)
-144  -$090  NewLayerInfo ()
-150  -$096  DisposeLayerInfo (a0:li)
-156  -$09c  FattenLayerInfo (a0:li)
-162  -$0a2  ThinLayerInfo (a0:li)
-168  -$0a8  MoveLayerInFrontOf (a0:movelayer,a1:backlayer)
-174  -$0ae  InstallClipRegion (a0:layer,a1:region)
```

H.19 Die MathFFP-Library (alphabetisch)

```
 -54  -$036  SPAbs (d0:float)
 -66  -$042  SPAdd (d1:leftFloat,d0:rightFloat)
 -96  -$060  SPCeil (d0:float)
 -42  -$02a  SPCmp (d1:leftFloat,d0:rightFloat)
 -84  -$054  SPDiv (d1:leftFloat,d0:rightFloat)
 -30  -$01e  SPFix (d0:float)
 -90  -$05a  SPFloor (d0:float)
 -36  -$024  SPFlt (d0:integer)
 -78  -$04e  SPMul (d1:leftFloat,d0:rightFloat)
 -60  -$03c  SPNeg (d0:float)
 -72  -$048  SPSub (d1:leftFloat,d0:rightFloat)
 -48  -$030  SPTst (d0:float)
```

H.20 Die MathFFP-Library (nach Offsets)

```
 -30  -$01e  SPFix (d0:float)
 -36  -$024  SPFlt (d0:integer)
 -42  -$02a  SPCmp (d1:leftFloat,d0:rightFloat)
 -48  -$030  SPTst (d0:float)
 -54  -$036  SPAbs (d0:float)
 -60  -$03c  SPNeg (d0:float)
 -66  -$042  SPAdd (d1:leftFloat,d0:rightFloat)
 -72  -$048  SPSub (d1:leftFloat,d0:rightFloat)
 -78  -$04e  SPMul (d1:leftFloat,d0:rightFloat)
 -84  -$054  SPDiv (d1:leftFloat,d0:rightFloat)
 -90  -$05a  SPFloor (d0:float)
 -96  -$060  SPCeil (d0:float)
```

H.21 Die MathIEEEDoubBas-Library (alphabetisch)

```
 -54  -$036  IEEEDPAbs (d0/d1:double)
 -66  -$042  IEEEDPAdd (d0/d1:double,d2/d3:double)
 -96  -$060  IEEEDPCeil (d0/d1:double)
 -42  -$02a  IEEEDPCmp (d0/d1:double,d2/d3:double)
 -84  -$054  IEEEDPDiv (d0/d1:double,d2/d3:double)
 -30  -$01e  IEEEDPFix (d0/d1:double)
 -90  -$05a  IEEEDPFloor (d0/d1:double)
 -36  -$024  IEEEDPFlt (d0:integer)
 -78  -$04e  IEEEDPMul (d0/d1:double,d2/d3:double)
 -60  -$03c  IEEEDPNeg (d0:integer)
 -72  -$048  IEEEDPSub (d0/d1:double,d2/d3:double)
 -48  -$030  IEEEDPTst (d0:integer)
```

H.22 Die MathIEEEDoubBas-Library (nach Offsets)

```
 -30  -$01e  IEEEDPFix (d0/d1:double)
 -36  -$024  IEEEDPFlt (d0:integer)
 -42  -$02a  IEEEDPCmp (d0/d1:double,d2/d3:double)
 -48  -$030  IEEEDPTst (d0:integer)
 -54  -$036  IEEEDPAbs (d0/d1:double)
 -60  -$03c  IEEEDPNeg (d0:integer)
 -66  -$042  IEEEDPAdd (d0/d1:double,d2/d3:double)
 -72  -$048  IEEEDPSub (d0/d1:double,d2/d3:double)
 -78  -$04e  IEEEDPMul (d0/d1:double,d2/d3:double)
 -84  -$054  IEEEDPDiv (d0/d1:double,d2/d3:double)
 -90  -$05a  IEEEDPFloor (d0/d1:double)
 -96  -$060  IEEEDPCeil (d0/d1:double)
```

H.23 Die MathTrans-Library (alphabetisch)

```
-120  -$078  SPAcos (d0:float)
-114  -$072  SPAsin (d0:float)
 -30  -$01e  SPAtan (d0:float)
 -42  -$02a  SPCos (d0:float)
 -66  -$042  SPCosh (d0:float)
 -78  -$04e  SPExp (d0:float)
-108  -$06c  SPFieee (d0:integer)
 -84  -$054  SPLog (d0:float)
-126  -$07e  SPLog10 (d0:float)
 -90  -$05a  SPPow (d1:leftFloat,d0:rightFloat)
 -36  -$024  SPSin (d0:float)
```

```
 -54  -$036  SPSincos (d1:leftFloat,d0:rightFloat)
 -60  -$03c  SPSinh (d0:float)
 -96  -$060  SPSqrt (d0:float)
 -48  -$030  SPTan (d0:float)
 -72  -$048  SPTanh (d0:float)
-102  -$066  SPTieee (d0:float)
```

H.24 Die MathTrans-Library (nach Offsets)

```
 -30  -$01e  SPAtan (d0:float)
 -36  -$024  SPSin (d0:float)
 -42  -$02a  SPCos (d0:float)
 -48  -$030  SPTan (d0:float)
 -54  -$036  SPSincos (d1:leftFloat,d0:rightFloat)
 -60  -$03c  SPSinh (d0:float)
 -66  -$042  SPCosh (d0:float)
 -72  -$048  SPTanh (d0:float)
 -78  -$04e  SPExp (d0:float)
 -84  -$054  SPLog (d0:float)
 -90  -$05a  SPPow (d1:leftFloat,d0:rightFloat)
 -96  -$060  SPSqrt (d0:float)
-102  -$066  SPTieee (d0:float)
-108  -$06c  SPFieee (d0:integer)
-114  -$072  SPAsin (d0:float)
-120  -$078  SPAcos (d0:float)
-126  -$07e  SPLog10 (d0:float)
```

H.25 Die PotGo-Library (alphabetisch)

```
  -6  -$006  AllocPotBits (d0:bits)
 -12  -$00c  FreePortBits (d0:bits)
 -18  -$012  WritePotGo (d0:word,d1:mask)
```

H.26 Die PotGo-Library (nach Offsets)

```
  -6  -$006  AllocPotBits (d0:bits)
 -12  -$00c  FreePortBits (d0:bits)
 -18  -$012  WritePotGo (d0:word,d1:mask)
```

H.27 Die Timer-Library (alphabetisch)

```
-42  -$02a  AddTime (a0:dest,a1:src)
-54  -$036  CmpTime (a0:dest,a1:src)
-48  -$030  SubTime (a0:dest,a1:src)
```

H.28 Die Timer-Library (nach Offsets)

```
-42  -$02a  AddTime (a0:dest,a1:src)
-48  -$030  SubTime (a0:dest,a1:src)
-54  -$036  CmpTime (a0:dest,a1:src)
```

H.29 Die Translator-Library (alphabetisch)

```
-30  -$01e  Translate (a0:input,d0:inputLength,a1:outBuf,d1:bufSize)
```

H.30 Die Translator-Library (nach Offsets)

```
-30  -$01e  Translate (a0:input,d0:inputLength,a1:outBuf,d1:bufSize)
```

AMIGA ASSEMBLER VON NULL AUF HUNDERT

Assembler – eine Programmiersprache deren Anwendungsvielfalt beispielhaft ist, findet auch auf dem Amiga immer mehr Freunde. Zu Unrecht wird diese Sprache als schwer erlernbar bezeichnet.

Das beiliegende Buch soll dieses Vorurteil ausräumen. Anhand von unzähligen Beispielprogrammen wird der Anfänger Schritt für Schritt mit der Sprache vertraut gemacht.

Für den fortgeschrittenen Programmierer ist das Buch ein unentbehrliches Nachschlagewerk.

Aus dem Inhalt:

- Syntax des MC 68000
- Datei-Operationen
- Arbeiten mit DOS-Fenstern
- Programmieren von Gadgets, Menüs, Requester...
- Laden, Ausgeben und Scalieren von Schriftsätzen
- Unterschiedlichste Grafikfunktionen
- Speicherverwaltung
- Multitasking
- Konstruktion eigener Libraries und Devices
- Die unterschiedlichen Devices
- DOS für Fortgeschrittene
- CLI-Schnellkurs
- Fehlermeldungen und ihre Bedeutung
- Tabelle aller Library-Routinen
- Unterschiede zwischen den Assemblern

Hardwarevoraussetzung:
alle Amiga von Commodore

Softtwarevoraussetzung:
Assemblerprogramm: z. B. DevPac, Seka, Profimat, OMA

Verlag Gabriele Lechner
Am Klostergarten 1
81241 München

ISBN 3-926858-38-40-0
DM 98,– sFr. 91,– öS 834,–